UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

DIVERSE INTRUSION-TOLERANT DATABASE REPLICATION

Paulo Jorge Botelho Ferreira

MESTRADO EM SEGURANÇA INFORMÁTICA

2012

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

DIVERSE INTRUSION-TOLERANT DATABASE REPLICATION

Paulo Jorge Botelho Ferreira

Orientador
Alysson Neves Bessani

MESTRADO EM SEGURANÇA INFORMÁTICA

2012

Resumo

A combinação da replicação de bases de dados com mecanismos de tolerância a falhas bizantinas ainda é um campo de pesquisa recente com projetos a surgirem nestes últimos anos. No entanto, a maioria dos protótipos desenvolvidos ou se focam em problemas muito específicos, ou são baseados em suposições que são muito difíceis de garantir numa situação do mundo real, como por exemplo ter um componente confiável.

Nesta tese apresentamos DivDB, um sistema de replicação de bases de dados diverso e tolerante a intrusões. O sistema está desenhado para ser incorporado dentro de um driver JDBC, o qual irá abstrair o utilizador de qualquer complexidade adicional dos mecanismos de tolerância a falhas bizantinas. O DivDB baseia-se na combinação de máquinas de estados replicadas com um algoritmo de processamento de transações, a fim de melhorar o seu desempenho. Para além disso, no DivDB é possível ligar cada réplica a um sistema de gestão de base de dados diferente, proporcionando assim diversidade ao sistema.

Propusemos, resolvemos e implementamos três problemas em aberto, existentes na conceção de um sistema de gestão de base de dados replicado: autenticação, processamento de transações e transferência de estado. Estas características torna o DivDB exclusivo, pois é o único sistema que compreende essas três funcionalidades implementadas num sistema de base de dados replicado.

A nossa implementação é suficientemente robusta para funcionar de forma segura num simples sistema de processamento de transações online. Para testar isso, utilizou-se o TPC-C, uma ferramenta de benchmarking que simula esse tipo de ambientes.

Palavras-chave: Bases de Dados, Middleware, Replicação , Diversidade, Tolerância a intrusões

Abstract

The combination of database replication with Byzantine fault tolerance mechanism is a recent field of research with projects appearing in the last few years. However most of the prototypes produced are either focused on very specific problems or are based on assumptions that are very hard to accomplish in a real world scenario (e.g., trusted component).

In this thesis we present DivDB, a Diverse Intrusion-Tolerant Database Replication system. It is designed to be incorporated inside a JDBC driver so that it abstracts the user from any added complexity from Byzantine Fault Tolerance mechanism. DivDB is based in State Machine Replication combined with a transaction handling algorithm in order to enhance its performance. DivDB is also able to have different database systems connected at each replica, enabling to achieve diversity.

We proposed, solved and implemented three open problems in the design of a replicated database system: authentication, transaction handling and state-transfer. This makes DivDB unique since it is the only system that comprises all these three features in a single database replication system.

Our implementation is robust enough to operate reliably in a simple Online Transaction Processing system. To test that, we used TPC-C, a benchmark tool that simulates that kind of environments.

Keywords: Databases, Middleware, Replication, Diversity, Intrusion Tolerance

Acknowledgments

There are several people I would like to thanks. Without their presence and help I would have not been able to do this thesis.

First, I would like to thank my advisor, Alysson Bessani for his guidance, patience and for keeping me motivated the whole time. I would also like to thank him for reviewing my thesis and helping me fix my mistakes.

To Marcel Santos, who without any kind of obligation also helped in reviewing my thesis.

My colleagues from the master program, for keeping me sane and made this journey so much easier to bear.

My friends and family, who have been allways there when I needed them.

Lisboa, Abril 2012

Dedicated to my friends and family, specially to my grandmother Lurdes, whose funeral I could not attend.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Every organization, no matter how small it is, have continuous information and data to store. However, the manipulation of this information has become impossible to be performed manually, this is mainly due to the fact that it is very time consuming (due to cataloguing data) and it is particularly susceptible to errors, since the operator is human and it eventually gets tired and its efficiency decreases. Therefore, it becomes easier to find information in a database which has become one of the most successful and reliable information technologies. Basically, the databases extend the function of the paper to store information on computers. Any company wishing to ensure effective control over its entire business, must have recur to management systems databases.

A DBMS (Database Management System) is nothing more than a set of applications that allow the users and companies to store, modify and extract information from a database. There are many different types of DBMS. From small systems that run on personal computers to huge systems that are associated with mainframes. A DBMS involves creating and maintaining databases, eliminating the need for specification of data definition, acts as an interface between application programs and data files and separates the physical and logical views of design data.

DBMS vendors in order to increase the robustness of their systems, they incorporate replication techniques. Many DBMS still do have this features. However all the replication does add is redundancy, which only allowed them to protect their systems against omission failures (e.g., crash failures, fail to receive a request, or fail to send a reply). However, with the technological advancements, threats on the area of networking and internet increased rapidly. These solutions are not sufficient if one really wants to achieve higher levels of security. A better solution is needed.

1.2 Motivation

So, with all the new technological advancements DBMS have become widely used across companies and corporations. However, the information that these companies store in their databases could be worth a lot of money. This also attracted a lot of malicious users. Before, if an malicious person wanted to steal any information, it would have to break in to steal the information. Now, the concept is still the same, but instead of breaking in a physical place, it has to break in the company network and then break in the mainframe that hosts the company database. Then the attacker has to find a vulnerability the DBMS and exploit it in order to the able to extract and steal information. This then lead to a lot of competition among DBMS vendors because even though performance is important, so is security, mainly for bigger companies, where a breach could cost millions. However, humans are not perfect beings, so logically, the software we produce is not 100% perfect as well. It is no mistake to say that, no matter how well designed or implemented a system is, it is very likely that is has some vulnerabilities (specially systems as complex as DBMS).

Then the concept of fault tolerance came up. Unlike traditional security, fault tolerance does not focuses on preventing faults, but instead in tolerating them, while having the system operating normally. That is the core idea of Byzantine Fault Tolerance (BFT), which came after fault tolerance. BFT not only protects systems from omission failures, like fault tolerance also does, but it also aims at tolerating Byzantine faults, which are basically arbitrary faults. BFT systems became very popular because they allowed a system to operate even in the presence of malicious faults.

A downside of BFT is that no protocol was efficient enough to be used in practice. The costs in performance simply did not make the scalable enough to be used. However, with the publication of PBFT [2], a State Machine Replication, algorithm things changed. PBFT algorithms and optimizations allowed BFT algorithms to become much more viable to be used.

There have been some research in the past few years about implementing BFT replicated database systems. There have been developed some prototypes that attempt so solve some problems related with BFT database replication[5, 3]. However, those prototypes were either only focused on very specific problems or had assumptions hard to fulfil (e.g., trusted component). In [5], very interesting algorithms proposed that take advantage of BFT to implement transaction handling in a BFT database replicated system.

However, even with the development of a BFT database replication system, it still does not prevent it from intrusions [6], which are malicious faults that result from a successful attacks from a malicious user. If an attacker compromises one of the replicas, he can also do the same for every other (if the faults are related to the system implementation or the database being used).This is where diversity comes into play. By taking advantage of diversity, one can use a different DBMS in each replica, and thus making the system intrusion tolerant.

All these research provide helpful knowledge that know can use and combine to generate new ideas and develop new systems with higher levels of security.

1.3 Contribution

In this thesis present DivDB, a Diverse Intrusion-Tolerant Database Replication system, composed by a JDBC Driver, the BFT-SMaRt replication Library and 3f+1 replicas with a different DBMS in each replica thus achieving diversity. This way we solve the problem of breaching a single database, since for an attacker to compromise the overall system it would have to maliciously affect f+1 replicas. In a diversified environment this is even more difficult since different DBMS are not likely to have common bugs.

We also found three specific problems related to BFT database replication. We solved the authentication process to the replicas through a JDBC interface. We implemented an algorithm for the transaction handling and also deployed a state-transfer solution. So, DivDB is unique in the sense it is the only system that combines all these features in a single prototype.

1.4 Organization

The remaining chapters in this thesis are structured in the following way:

In Chapter 2, we will introduce the main concepts to help the reader be contextualized about what will be discussed later. Besides that we also present some related work that has been done in this area of research.

In Chapter 3 we will go through our system implementation, by discussing the problems we approached and the solutions we came up with. After that we present our architecture implementation.

In Chapter 4 we explain TPC-C, the benchmarking tool used. We define our test environment and present the some performance results for DivDB.

Finally at Chapter 5, The thesis is concluded with an overview of what has been done and point out possible fields to further research in the future.

Chapter 2

Contextualization and Related Work

In this section we will try to explain all the required concepts and background information that will help understand what is to come further on. We will start by giving a contextualization on databases and explains its most relevant mechanisms for this subject.

Then we will move into Intrusion Tolerance where we will mention essential concepts and protocols that will help understand what we did and how we did it.

Finally, we will also present some related literature of similar projects that dealt with some of the problems we also had to take care of, namely transaction handling.

2.1 Databases

The notion of a database [7, 8, 9] is nothing more than a collection of information that exists over a long period of time. The essence of databases comes from a set of knowledge and technology that has been developed over the years and came to life as a specialized software that we call DBMS (Database Management System). A DBMS is a versatile and powerful tool that enables one to create and manage large amounts of data in an efficient way. It also allows the information to be persistently stored for large periods of time and to be transferred from one place to another much easier way than any traditional information storage, such as shelves of files and books.

The functionalities provided by a DBMS can range from creating databases and specifying their schema (logical structure of the data) to the ability of querying and modifying the data inside the databases. This manipulation of the data inside a database is done through the use of an appropriated language, usually called query language or data-manipulation language (DML). This kind of language is a family of syntax elements similar to a computer programming language used for inserting, deleting and updating data in a database.

2.1.1 SQL

The Structured Query Language (SQL) [9, 7] is very known and popular data-manipulation language. It can be used to retrieve and manipulate data in relational databases. This concept of relational databases system first appeared in [10] where it was proposed that database system should present the user with a view of data organized as tables called relations. Ideally there should be a complex data structure that would allow rapid response to the queries being made. All these queries made to the databases would be done using SQL language which would greatly increase the efficiency of database programmers.

The most common interaction with a database is the SELECT statement, which is an element of SQL. The SELECT statement allows an operator to dig though one or more database tables and display just the data that meets specific criteria. Other frequently used statements include INSERT, for inserting new data into a table, UPDATE, for modifying existing data in a table; and DELETE for removing rows.

2.1.2 Transactions

Transactions can be seen as a set of queries and other SQL operations that are agglomerated in one unit that must be executed atomically and isolated from one another. Besides that, the effect of any completed transaction must be preserved even if in the presence of a system failure right after the competition of the transaction.

Normally, a DBMS is equipped with a set of functionalities such as the ability to control data access, allowing the operator to mediate the access to the contents inside the databases and what can be done with it. It is also enforces data integrity, in order to ensure that the information is correct and consistent within the database. DBMS also possess concurrency control mechanisms mainly to ensure that transactions processing is done correctly. And finally, it not only enables database recovery after failures and restore it from backup files, but it also has mechanisms to maintain database security.

2.1.2.1 ACID Properties

For a transaction to be valid, it must respect a set of properties. A correct transaction must be atomic, consistent, isolated and durable [7, 11, 12]. These four properties are called by the acronym ACID. If database transactions meet these properties and the concurrency-control manager also respects them, then we know that all transactions that are processed in a given database are valid.

Going in more detail through each property, we can see that:

- **Atomicity** requires that database modifications must follow an "all or nothing" rule. More precisely, if one operation of the transaction fails, then the entire transaction will fail and the database state must be rolled back to before executing the transaction. Even in the presence of a failure, error or crash, the database must guarantee the atomicity property in each and every situation. This also ensures that an unfinished transaction cannot exist.

- **Consistency** property ensures that any transaction done will take the database from one valid state into another valid state. Consistency states that only consistent (valid according to all the rules defined) data will be written to the database. In other words, any rows that are affected by the transaction will remain consistent with any operation that is applied to them. This consistency definition applies to everything that is modified by the operations inside the transaction.
- **Isolation** ensures that no transaction should be able to interfere with any other transaction execution. In other words, two running transactions should not be able to affect the same rows. If this was possible, the outcome would be unpredicted and the system unreliable. However, this property can be relaxed (i.e., partly respected) because of the huge speed decrease this type of concurrency management implies. In fact, the only way to strictly respect this property is to use a serializable model where no two transactions can affect the same data at the same time and where the result is predictable.
- **Durability** means that once a transaction has been committed, it will remain so. So, every change made to the information before the commit is protected against power loss / crash / errors and cannot be lost by the system.

Moving on to transaction processing, in order to process transactions, DBMS recur to a concurrency-control manager, or scheduler, which is responsible for ensuring the atomicity and isolation of transactions. Although transactions must appear to be executing in isolation, in most systems, there might be more than one transaction executing at once. This is due to the fact that in order to achieve higher performance, systems must allow to run concurrent transaction so they can deal with large amount of requests simultaneously. However, this is very likely to lead to concurrency problems. The scheduler main task is to deal with these issues. For example, it is the scheduler job to ensure that the individual operations of different transactions are executed in such an order that the overall effect is the same as if the transaction had in fact executed one-at-a-time.

2.1.3 Stored Procedures

A stored procedure [12, 7] can be seen as a subroutine available to the application that can execute a set of commands over a database system. Stored procedures are quite similar to a function call in a programming language, they can accept parameters and also return a value or a result set. The concept of stored procedures was first officially introduced in the SQL standard SQL2003. It was specified to allow the user to write these procedures in a simple, general-purpose language and to store them inside the database, as part of the schema. Stored procedures not only could offer other functionalities that SQL by itself could not perform, but they also helped to consolidate and centralize the logic that was originally implemented in applications all inside the procedure itself. So, any extensive or complex processing that requires the execution of several SQL statements can be moved into stored procedures to be executed in the database server. Besides that they also allow to have nested procedures.

2.1.4 Concurrency Control

Concurrency Control is an important component of DBMS since it helps ensure that database transactions are performed concurrently and without violating the integrity of the database itself.

The DBMS execute each operation atomically. So, this means that the DBMS behaves as if it executes the operations sequentially, that is, one at a time. In order to achieve this behaviour, the most simple approach is to actually execute the operations sequentially. However, in order to achieve greater performance they might also execute operations concurrently. In other words, there may be times when it is executing more than one operation at once. However, even if operations are executed concurrently, the final effect must be the same as sequential execution.

It is important to note that the interactions among concurrently executing transactions can cause the database state to become inconsistent, even when the transactions individually preserve correctness of the state, and there is no system failure. Therefore, when talking isolation in transactions, serializability is an very important concept when it comes to concurrent transaction handling.

2.1.4.1 Serializability

The definition of **serializability** [7, 11] in transaction processing stands that a transaction schedule is serializable if its outcome (e.g., the resulting database state) is equal to the outcome of its transactions executed serially, sequentially without overlapping in time. Normally, transactions are executed concurrently, since this is the most efficient way. However, serializability is the major correctness criterion for concurrent transactions execution. It is considered the highest level of isolation between transactions, and plays an essential role in concurrency control. As such it is supported in all general purpose database systems.

2.1.4.2 Categories

In practice, the methods used in concurrency control to mediate the access to the database information can be classified in two types:

- **Pessimistic concurrency control**
  - A system of locks prevents users from modifying data in a way that affects other users. After a user performs an action that causes a lock to be applied, other users cannot perform actions that would conflict with the lock until the owner releases it. Note that these locks are only applied to certain tables or rows, not to the entire database. This is the so called pessimistic control because it is mainly used in environments where there is high contention for data, where the cost of protecting data with locks is less than the cost of rolling back transactions if concurrency conflicts occur.

- **Optimistic concurrency control**

  - In optimistic concurrency control, users do not lock data when they read it. When a user updates data, the system checks to see if another user changed the data after it was read. If another user updated the data, an error is raised. Typically, the user receiving the error rolls back the transaction and starts over. This is called optimistic because it is mainly used in environments where there is low contention for data, and where the cost of occasionally rolling back a transaction is lower than the cost of locking data.

Different categories provide different performance, for example, different average transaction completion rates, depending on transaction types mix, computing level of parallelism, and other factors. If selection and knowledge about trade-offs are available, then category and method should be chosen to provide the highest performance.

Typically, most of commercial DBMS use by default a pessimistic concurrency control. This is due to the fact that these DBMS are built to support huge loads concurrent transactions processing and without the use of locks a lot of rollbacks could happen.

2.1.5 Multiversion concurrency control

Multiversion concurrency control (MVCC) [13] is a concurrency control method commonly used by DBMS to provide concurrent access to the databases and in programming languages to implement atomic operations. With MVCC, the updates on a database data are done not by deleting the old data and replacing it with the new one, but instead by marking the old data as obsolete and adding the newer version. Thus there are multiple versions stored, but only one is the latest.

MVCC also provides potential point in time consistent views. In fact read transactions under MVCC typically use a timestamp or transaction ID to determine what state of the DB to read, and read these versions of the data. This avoids managing locks for read transactions because writes can be isolated by virtue of the old versions being maintained, rather than through a process of locks or mutexes. Writes affect future version but at the transaction ID that the read is working at, everything is guaranteed to be consistent because the writes are occurring at a later transaction ID. In other words, MVCC provides to each connection to the database with a snapshot of the database for that connection to work with. Any changes made will only be seen by other connections after the transaction has been committed

2.1.5.1 Snapshot Isolation

Snapshot isolation (SI) [5, 14] arose from work on multi-version concurrency control databases, where multiple versions of the database are maintained concurrently to allow readers to execute without colliding with writers. In practice snapshot isolation is implemented within MVCC, where multiple versions of each data object are maintained.

The concept of snapshot isolation semantics can be seen as a guarantee that all reads made in a transaction will see a consistent snapshot of the database. The transaction itself will successfully commit only if no updates it has made conflict with any concurrent updates made since that snapshot.

The main reason for its adoption is that it allows better performance than serializability, yet still avoids most of the concurrency anomalies that serializability avoids.

2.1.6 JDBC

The Java DataBase Connectivity, commonly referred to as JDBC [15], is an industry standard API for database-independent connectivity between the Java programming language and a wide range of databases. It defines how a client may access a database and it provides methods for querying and updating data in a database.

2.1.6.1 JDBC Architecture

The JDBC API can be built to support database access through a two-tier or three-tier processing model [16], as shown in Figure 2.1.

In the two-tier architectural model, a Java application talks directly to the database. This requires that the appropriated JDBC driver is used so the application can communicate with the particular DBMS being accessed. The commands send by the application are delivered to the database, and the results of those statements are sent back to the user. The database may be located on any machine connected through the network to the user machine. So, this is referred to as a client/server configuration, where the user machine acts as the client, and the machine hosting the database as the server. The network can either be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

In the three-tier model illustrated in Figure 2.2, the client machine no longer communicates directly with the database, but instead the commands are sent to a ”middle tier” of services, which then sends the commands to the database. The DBMS processes the commands and sends the results back to the middle tier, which then sends them to the user. This three-tier model is very attractive in business environments because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages since it splits the tier of logistics and storage into two different layers.

In both processing models the application that communicates with the DBMS has to do it through the JDBC API. The JDBC API structure can be seen in two different perspectives. The first is the JDBC API for application writers, and the second is the lower-level JDBC driver API for driver writers. Regarding application writers, all they have to do is interact with the JDBC API interface in order to access the database and operate with it.

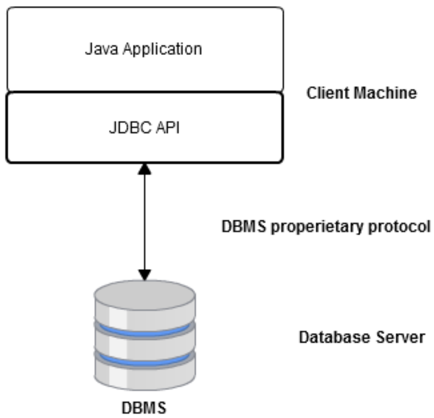

Figure 2.1: 2-Tier Architecture

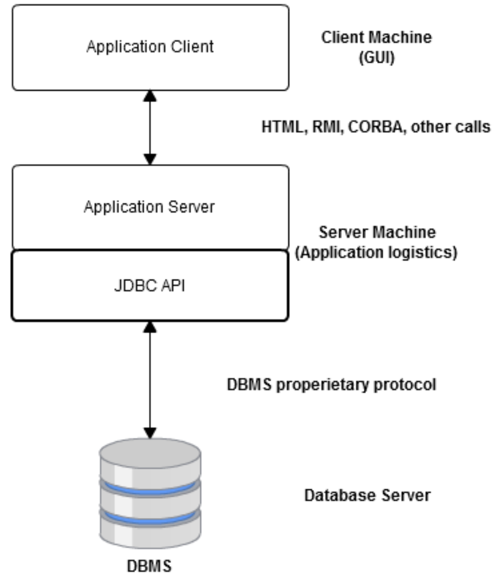

Figure 2.2: 3-Tier Architecture

2.1.6.2 JDBC Drivers

Now, as for JDBC drivers [17, 18], they are the software component that enable a Java application to indeed interact with the database. In order to communicate with an individual database, a specific JDBC driver is required to communicate with it. Each JDBC driver gives out the connection to that specific database and implements the protocol that is used to transfer queries and result sets between the client applications and the database itself.

There are several kinds of JDBC drivers which can either be proprietary or free. We will discuss each one of them and try to point out the pros and cons of each of one of them. They can be categorized according to their functionality into one of the following types.

Type 1 driver - JDBC <-> ODBC Bridge

The JDBC type1, which is also known as the JDBC-ODBC bridge (see Figure 2.3), can be seen as a database driver implementation that uses the ODBC driver to connect to the database. What the driver basically does is to converts JDBC method calls into ODBC function calls.

An ODBC [19] (Open Database Connectivity) driver defines a standard C API for accessing a relational DBMS. In other words, ODBC provides a universal middleware layer between the application and DBMS, allowing the application developer to use a single interface so that if changes are made to the DBMS specification, only the driver needs to be updated.

The main function of this kind of driver is to translate the queries obtained by JDBC into corresponding ODBC query, which is then handled by the ODBC driver. This makes this kind of driver easy to connect. However, in terms of performance, there is an overhead since the queries have to go through the JDBC<->ODBC bridge to reach the ODBC driver, then to the native database connectivity interface. Besides that, the ODBC has to be installed on the client machine that communicates with the DBMS, which can lead to some compatibility problems since ODBC is platform dependent.

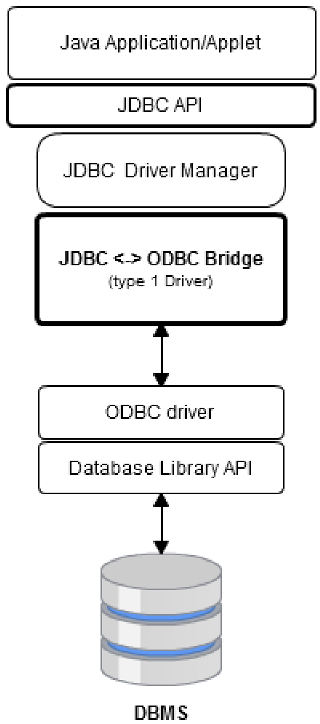

Figure 2.3: Type 1 driver

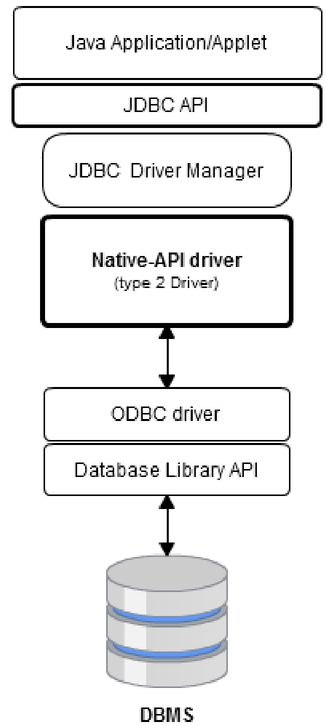

Figure 2.4: Type 2 driver

Type 2 Driver - Native-API Driver

The type 2 driver, typically known as the Native-API driver or Partial Java driver, is an implementation of a database driver that uses the client-side libraries of the database. Its representation is shown in Figure 2.4.

The driver directly converts JDBC method calls into native calls of the database API. This kind of driver is not entirely written in Java since is connected with non-Java code that makes the final database calls. Type 2 drivers are specifically compiled to be used with a particular operating system. In other words, type 2 drivers, like type 1, are platform dependent.

Type 3 Driver - Network-Protocol Driver

The JDBC type 3 driver is a database driver implementation which makes use of a middle tier between the calling program and the database. Because of this middle tier, this driver is also known as the Pure Java Driver for Database Middleware. So, the middleware (application server) has the task of converting JDBC calls directly or indirectly into the vendor-specific database protocol, depending on the database it is communicating with.

The driver is entirely written in Java and the same driver can be used to connect to multiple databases. The type 3 drivers are platform-independent since the platform-related differences can be dealt by the middleware. Also, making use of the middleware provides additional advantages of security and firewall access.

We can clearly see on Figure 2.5 that type 3 drivers follow a three-tier communication architecture. Besides that, they can communicate with multiple databases (the databases do not have to be vendor specific).

Then main advantages are:

- The communication between the client and the middleware server is database independent since the Network-protocol driver relieves the communication from any specifying details of each database being used.
- Any changes made to a database does not affect the client at all since the Middleware Server abstracts the client from all the specific details of the databases it communicates with.
- The Middleware Server can provide typical middleware services like caching (connections, query results, and so on), load balancing, replication, logging, auditing etc.

As for its disadvantages:

- The middleware server requires database-specific capabilities in its code for each different database it has to connect to.
- The middleware server also becomes the bottleneck of the whole system since all the information will have to pass by him and be converted into the appropriated format.

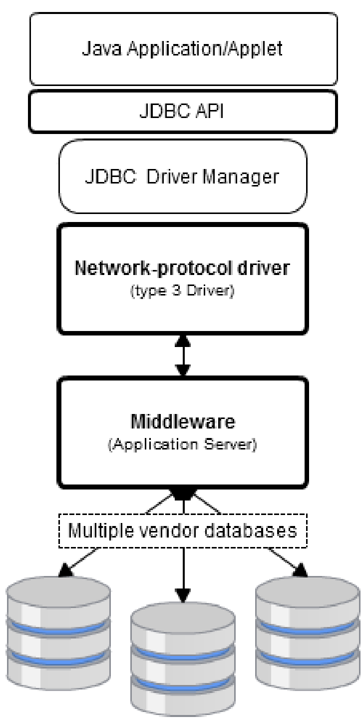

Figure 2.5: Type 3 driver

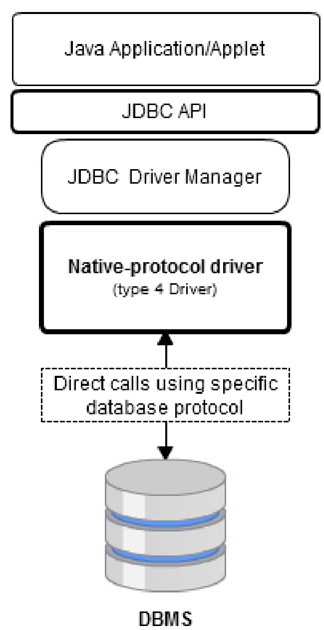

Figure 2.6: Type 4 driver

Type 4 Driver - Native-Protocol Driver

The JDBC type 4 driver, also known as the Direct-to-Database Pure Java Driver, is a database driver in which the JDBC calls are converted directly into a vendor-specific databases protocol. It is represented in Figure 2.6.

Typically, these driver implementations are written completely in Java, and therefore they are platform independent. They tend to have a better degree of performance than the type 1 and type 2 drivers since they do not the overhead of converting calls into ODBC or database API calls. Unlike the type 3 drivers, it does not need associated software to work. As the database protocol is vendor-specific, the JDBC client requires separate drivers, usually vendor-specific drivers connect to different types of databases.

Summing up, the main advantages of type 4 drivers are:

- They are entirely implemented in Java, and therefore granting platform independence. Besides that, it does not need any kind of translation or middleware layers, freeing it from any kind of extra overhead.
- The driver converts JDBC calls into vendor-specific database protocol so that client applications can communicate directly with the database server.

As for downsides of this driver, the main problem a user can have is the need to have the drivers for each database it wants to connect to. This is due to the fact that drivers are database dependent.

2.2 Intrusion Tolerance

Intrusion Tolerance (IT) started to emerge as a new approach in the last decade and gained a significant importance in the area of fault tolerance due to the need of higher degree of security and reliability nowadays. The notion of IT is that a system is able to tolerate a wide set of fault, agglomerating both malicious and non-malicious faults. In other words, the idea is to be able to handle the faults so that they do not lead to the failure of the system.

Intrusion Tolerance differs from classical security in the aspect that instead of trying to prevent every single intrusion, what is done is to acknowledge that intrusions might occur, building the system to tolerate them. The system is able to trigger mechanisms that prevent the intrusion from affecting its overall operation.

2.2.1 AVI composite fault model

The sequence of events that lead to the failure of a component is what is called the AVI composite fault model [1]. Security-wise, these events are the faults that can lead to an intrusion, and therefore a component failure. These faults can either be attacks and vulnerabilities .

In order to properly understand the AVI fault model, it is important to explain some of the concepts that were just mentioned:

- **Fault:** A fault is a defect in a system. The presence of a fault in a system may or may not lead to a failure.
- **Error:** An error represents the difference between the intended behaviour of a system and its actual behaviour inside the system boundary.
- **Failure:** A failure is an instance in time when a system displays behaviour that is contrary to its specification.

The AVI fault model describes the way these faults occur by presenting the sequence: attack -> vulnerability -> intrusion.

So, the causes of an intrusion are:

Vulnerability: a fault in a computing or communication system that can be exploited by an attack.

Attack: an intentional attempt to maliciously exploit a fault in a computing or communication system.

These faults lead to **intrusions**, which are malicious operation faults that result from a successful attack on a vulnerability.

It is important to be able to differentiate each of the several kinds of faults that may lead to a security failure. Figure 2.7 (a) illustrates the relationship between attacks, vulnerabilities and intrusions. So basically, the sequence attack -> vulnerability -> intrusion -> failure defines what we so likely call the AVI composite fault model.

Vulnerabilities can be seen as the primary faults existing inside the components. So, all requirements, specification, design or configuration faults can be seen as vulnerabilities that a system has and they are typically accidental. Attacks, on the other hand, are interaction faults that maliciously attempt to activate one or more of those vulnerabilities.

If an attack successfully triggers a vulnerability, then an intrusion occurs. At this stage, the system normally enters an erroneous state. This is where Intrusion tolerance comes in, for example, the errors resulting from the intrusion can be discovered by using an intrusion detection mechanism, and they can be recovered or masked. However, if nothing is done to process the errors resulting from the intrusion, the failure of some or several security properties is most likely to occur.

2.2.1.1 The need for Intrusion Tolerance

The typical goal of most systems is to attempt to avoid the occurrence of intrusion faults (intrusion prevention). However, to be able to ensure perfect prevention is infeasible. It would require the system to handle not only all the currently known attacks, but also new attacks that might arise. So, for any intrusions that might escape the prevention process, there is the need for some form of intrusion tolerance in order to prevent the system failure even in the presence of intrusions.

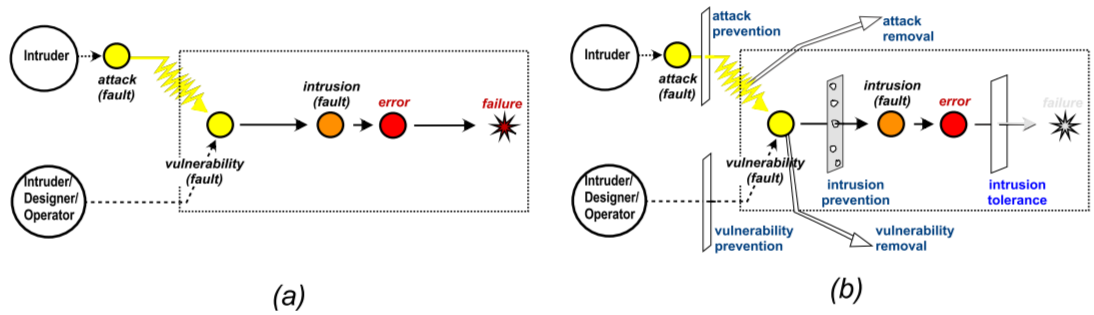

Figure 2.7: (a) AVI composite fault model; (b) Preventing security failure[1]

2.2.2 Byzantine Fault-Tolerance

Before starting with the concept of Byzantine fault tolerance, one must first understand the meaning of a Byzantine failure. So, the definition [20] of **Byzantine failure** states that a component of a system may fail in arbitrary ways. So, a Byzantine fault is an arbitrary fault that occurs during the execution of an application on a distributed system. It comprises both omission failures (e.g., crash failures, failing to receive a request, or failing to send a response) and commission failures (e.g., processing a request incorrectly, delayed clocks, corrupting local state, and/or sending an incorrect or inconsistent response to a request).

Byzantine fault tolerance is a sub-field of fault tolerance research which was inspired by the Byzantine Generals Problem [21]. Byzantine fault tolerance also plays an important role in the design of intrusion tolerant systems. The main objective of Byzantine fault tolerance is to be able to protect the system against Byzantine failures. Correctly functioning components of a Byzantine fault tolerant system will be able to correctly provide the system's service assuming that the number of Byzantine fault does not exceed a certain bound.

2.2.2.1 State Machine Replication

State Machine Replication [22] is a replication technique where an arbitrary number of clients send commands to a set of replicated servers. These servers, typically called by replicas, implement a stateful service that changes its state after processing client commands, and sends replies to the clients that issued them.

The main goal of this technique is to make the state at each replica to evolve in a consistent way, thus making the service completely and accurately replicated at each replica. In order to achieve this behaviour, it is necessary to satisfy three properties:

- Replicas only apply deterministic changes to the state;
- All replicas start with the same state;
- All replicas execute the same sequence of operations.

Regarding the first property, it is essential that each operation must be deterministic so that the same operation produces the same result in each different replica. If the operations could be non-deterministic, the replicas states would eventually diverge.

The second property can be easily accomplished if the setup and configuration of all replicas is done in the same manner.

Unlike the previous properties, the last one requires the replicas to communicate among themselves, through the execution of an agreement protocol, in order to guarantee that all the commands are executed in the same order across all replicas.

2.2.2.2 Practical Byzantine Fault Tolerance

Byzantine fault tolerant replication protocols used to be considered too expensive to be practical. Previous work required too many messages, it was based on public key cryptography and it assumed synchrony since it had bounds on message delays and process speeds. But then Miguel Castro and Barbara Liskov introduced the "Practical Byzantine Fault Tolerance" (PBFT) algorithm [2], which was aimed to provide high-performance Byzantine state machine replication, which enables replicas to process thousands of requests per second with sub-millisecond latency.

The System Model of this algorithm is defined as follows:

- Networks are unreliable, in other words, messages can be delayed, reordered, dropped or retransmitted.
- The adversary can co-ordinate faulty nodes or intentionally delay communication.
- The adversary cannot delay correct nodes or break cryptographic protocols.
- If an replica is Byzantine, then it may behave in an arbitrary way and it does not need to follow the protocol.
- Each replica can verify the authenticity of the messages sent to them.
- Total number of replicas is 3f+1, where f is the maximum number the system will be able to tolerate while still working properly.

Regarding the properties the system must fulfil, it must ensure:

- **Safety**: The system maintains state and appears to the client like a non-replicated remote service.
- **Liveness**: Correct clients will eventually receive a reply to every request sent. If a request from a correct client does not complete during the current view, then a view change occurs.

The concept of views within PBFT is very straightforward. Operations occur within views. For a given view, a particular replica is the primary node and the others are backup nodes.

Besides the normal operation mode, the PBFT algorithm also has a view-change mode. The first represents the full procedure from the client request until it receives the correct result from at least f+1 of the replicas. Figure 2.8 shows the operation of the algorithm in the normal case where there are no faults and replica 0 is the primary.

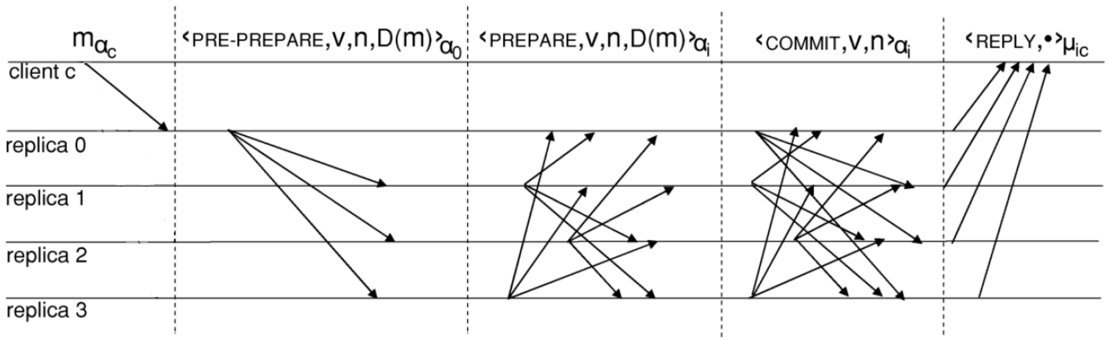

Figure 2.8: Normal Operation Mode[2]

1. **Pre-prepare:** The purpose of this phase is to establish a sequence number to the given message m. The primary picks the sequence number then sends it in a PRE-PREPARE message to the other replicas.
2. **Prepare:** Each replica will send a PREPARE message to each other. After receiving at least 2f PREPARE messages, they proceed to next phase.
3. **Commit:** Each replica multicasts a COMMIT message to each other. If a replica receives 2f COMMIT messages with the correct parameters and a valid sequence number, then the request is executed and the message is considered committed.
4. **Reply**: Each functioning replica sends a reply directly to the client. This bypasses the case where the primary fails between request and reply.

As for the second operation mode, it corresponds to the view-change protocol which provides liveness by allowing the system to make progress even when the primary fails. Figure 2.9 illustrates the view-change operation where the primary replica 0 is faulty.

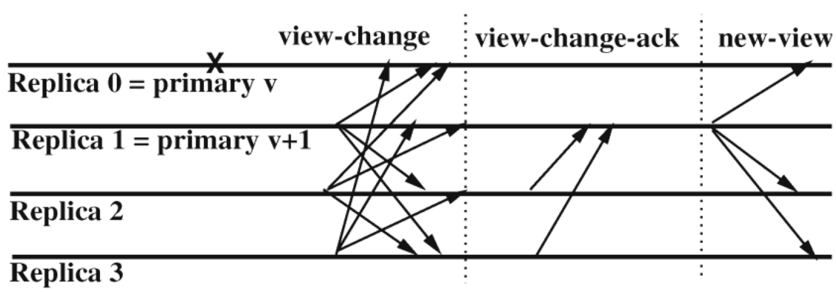

Figure 2.9: View-Change Mode[2]

1. **View-change**: A backup replica triggers the view change protocol if it stays with some pending message m for more than a certain time limit (request timeout expires);
2. **View-change-ack**: The replicas send a view-change-ack to the primary of the next view (v+1). Once the new primary receives 2f-1 view-change-acks messages, it will accept the view-change.
3. **New-view**: The new primary will then multicast a new-view message too all the remaining replicas.

For the algorithm to work correctly, the client needs to be BFT-aware. They must implement timeouts for view-change and must wait for replies directly from the replicas. Once the clients receive f+1 replies, the results are accepted and guaranteed correct, even if in the presence of f failed replicas.

2.2.2.3 Optimizations

The PBFT was designed to be practical and usable in real systems. For that reason the authors also developed several optimizations to the original algorithm they proposed. These optimizations improve the performance of the algorithm during normal-case operation while preserving the liveness and safety properties. We will mention here the most relevant to our context, database replication:

Symmetric Cryptography: The original PBFT algorithm uses asymmetric cryptography to authenticate all messages. However, they proposed an optimization where they would use digital signatures only for view-change and new-view messages, which are sent rarely. All other messages would be authenticated using message authentication codes (MACs). MACs can be computed three orders of magnitude faster than digital signatures, thus eliminating the main performance bottleneck in previous systems [23, 24].

Batching: The idea behind batching is that instead of running the agreement protocol for every request that is going to be executed, it is done to sets of requests. So a batch of requests is gathered then sent as one message. This way the agreement protocol is only executed once, over the batch of messages. This technique can be used to dramatically increase the protocol throughput.

2.2.3 Diversity

The use of Byzantine Fault Tolerance algorithms such as State Machine Replication plays an important role in the design of an Intrusion Tolerant System. However, the system is still vulnerable to what we call common mode faults. In other words, attacks can be cloned and directed automatically and simultaneously to all (identical) replicas.

This is where diversity comes in. It can be used to reduce the common-mode vulnerabilities among all replicas. For example, by using different operating systems in each replica we reduce the chance that they have common vulnerabilities and therefore the same attack will not work on all the replicas

[25]. It not only reduces the probability of common-mode vulnerabilities, but also the probability of common-mode attacks (by obliging the attacker to master attacks to more than one architecture) [26]

2.2.3.1 Database Diversity

Replication can be used as an effective intrusion tolerance mechanism if the multiple copies of the database do not usually fail together on the same operation, or at least they tend not to fail with identical erroneous results. For this purpose, different forms of diversity to be used in intrusion tolerant systems:

- Simple **separation of redundant executions**, which is the most basic form but still it can tolerate some non-deterministic failures since the same execution may not lead to the failure of all the databases.
- **Design diversity:** this form of diversity protects the overall system against design faults. The multiple replicas of the database are managed by different DBMS products, so like we have discussed before, the probability of having common-mode vulnerabilities is smaller.
- **Data diversity** [27]: by rephrasing statements into different but logically equivalent sequence to produce redundant executions. This gives a better chance of a failure not being repeated when the rephrased sequence is executed on another replica of even the same DBMS product. It was reported on [28] that a set of "rephrasing rules" that would tolerate at least 60% of the bugs examined in their studies.
- **Configuration diversity:** As we know, DBMS products have many configuration parameters. Some of these parameters can produce different implementations of the data and operation sequences. Therefore, this can be used to reduce the risk of the same bug being triggered in two DBMS from the same vendor but with different configurations.

These precautions can in principle be combined (for instance, data diversity can be used with diverse DBMS products), and implemented in various ways, including manual application by a human operator.

2.3 Related Work

This section provides an overview on current research in the area of intrusion-tolerant databases. I will describe different approaches that solved specific problems in the area database replication. These solution not only were aimed at solving these problems but they also improved the security of their solutions by applying Intrusion Tolerance mechanisms.

Some of the problems these solutions try to solve are the distributed transaction processing, leader election and an uniform representation of data retrieved from different DBMS.

2.3.1 BASE

The BASE acronym comes from BFT with Abstraction Specification Encapsulation and the idea of this approach is to be able to combine both Byzantine fault tolerance [29] with data abstraction [30] mechanisms. As it has been previously explained, Byzantine fault tolerance allows us to replicate a service in such way that it can tolerate arbitrary behaviour from faulty replicas. As for Abstraction, it hides implementations details to enable the reuse of off-the-self implementation of important services and to improve the ability to mask software errors.

Although off-the-shelf implementations of the same service offer pretty much the same functionality, they behave differently because they implement different specifications, using different representations of the service state. BASE, like any other state machine replication mechanism, requires determinism. If replicas that execute the same sequence of operations do not produce the same sequence of outputs, then the system will be in an inconsistent global state. So, what BASE does is define a common abstract specification.

This specification defines the abstract state, an initial state value and the behaviour of each service operation. Also, the specification is defined without knowledge of the internals of each implementation (enabling us to treat them as black boxes). The abstract state only needs to cover the information that is going to be presented to the clients. It does not have to mimic everything that is common among all the states from each different implementation.

Another important component of BASE Library is the conformance wrappers. These wrappers implement the common specification for each implementation. Basically they work as a sort of bridge between each implementation and the common specification. They are responsible for translating the concrete behaviour of each implementation to the abstract behaviour.

The last relevant component of BASE is called abstraction function (paired with one or more inverse functions). These abstraction functions allow state transfer among the replicas. Like explained before, state transfer is an important mechanism since it allows to repair faulty replicas and bring slow replicas up-to-date when messages are missing or have been garbage collected. The abstraction function is used to convert the concrete state stored by a replica into the abstract state which will be transferred to another replica. The replica that receives it, uses the inverse abstraction function in order to convert the abstract state into its own concrete state representation.

Although in theory this methodology can be used to build a replicated service from any set of existing implementations of any services, the truth is that sometimes it may not be so straightforward in practice because of the following problems:

- Undocumented behaviour: In order to be able to apply the methodology, we need to understand and model the behaviour of each service implementation.
- Very different behaviour: If the implementations used to build the service behave very differently, any common abstract specification will deviate significantly from the behaviour of some implementations.
- Limited interfaces: The external interface of some implementations may not allow the wrapping code to read or write data that has an impact on the behaviour observed by the client.

Although these problems might arise, the fact is that BASE presents some very interesting architectural ideas of how to develop a Byzantine fault tolerance database replicated service. Its abstraction mechanisms can be quite useful when applying diversity into such kind of systems which can help normalize any variations that different DBMS can have.

2.3.2 Heterogeneous Replicated DB (Commit Barrier Scheduling)

One of the main challenges in designing a replication scheme for transaction processing systems is ensuring that the different replicas execute transactions in equivalent serial orders while allowing a high degree of concurrency. Heterogeneous Replicated DB (HRDB) [3] scheme meets this goal by using a concurrency control protocol called commit barrier scheduling (CBS). This was done using unmodified production versions of several commercial and open source databases as replicas.

Some important features regarding HRDB design are:

- Databases use strict two-phase locking concurrency control mechanism. Strict two-phase locking transactions acquires read and write locks on data items as they access them, and these locks are held until the end of the transaction.
- Clients do not interact directly with the database replicas. Instead they communicate with a shepherd, which acts as a front-end to the replicas and coordinates them.
- Besides that, HRDB is parametrized by f, the number of simultaneously faulty replicas it can tolerate. So, it only requires 2f+1 database replicas because the replicas do not carry out agreement. They simply execute statements sent to them by the shepherd
- The shepherd is assumed to be trusted (although in a real environment this assumption might be hard to fulfil).

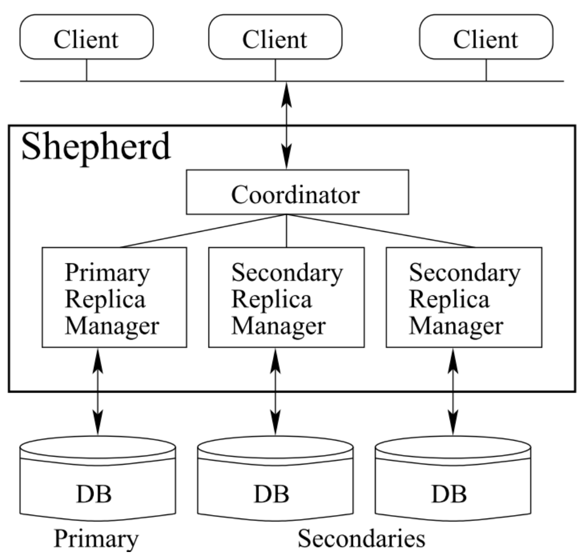

Figure 2.10: HRDB system architecture Vandiver et al. [3]

The **shepherd** agglomerates a set of components. As it can be seen in the Figure 2.10, inside it runs a single coordinator and one replica manager for each back-end replica.

The way things work inside the shepherd are as follows:

1. The coordinator receives statements from clients and forwards them to the replica managers.
2. Replica managers execute statements on their replicas, and send answers back to the coordinator.
3. The coordinator sends results back to the clients, compares query answers for agreement, and determines when it is safe for transactions to commit.

2.3.2.1 Commit Barrier Scheduling

In CBS, one replica is designated to be the primary, and runs statements of transactions slightly in advance of the other secondary replicas. CBS ensures that all non-faulty replicas with an equivalent logical state will return the same answer to any given query. This ensures that all non-faulty replicas execute committed transactions in equivalent serial orders.

CBS does not limit concurrency for processing queries at the primary in any way. So, when the coordinator receives a query from a client, it immediately sends it to the primary replica manager, which forwards it to the primary replica. Hence, the primary replica can process queries from many transactions simultaneously using its internal concurrency control mechanism (strict two-phase locking). As soon as it returns a response to a query, the coordinator sends that query to each secondary replica manager. Each of them adds the query to a pool of statements that will eventually be executed at the corresponding secondary.

If the primary is faulty, CBS may be unable to make progress efficiently. The way to handle a faulty primary is to do a view change in which a new primary is selected by the coordinator and the old one is demoted to being a secondary.

As far as the safety and liveness properties are cornered:

- **Safety** is ensured since it guarantees that correct replicas have equivalent logical state, and that clients always get correct answers to transactions that commit.
- **Liveness** is ensured mainly thanks to the view change mechanism and CBS.

2.3.3 Byzantium

Byzantium [5] is a BFT database replication middleware that provides snapshot isolation semantics. It allows concurrent transaction execution without relying on any centralized component. The main design features of Byzantium are:

- Snapshot isolation, where it guarantees that all reads made in a transaction will see a consistent snapshot of the database.

- Middleware-based replication, similar to a JDBC type 3 driver: the client communicates with a middleware application using a database independent protocol. The middleware translates the requests into the target database specific commands. This approach allows the use of existing databases without the need to modify them, it even allows distinct implementations from different vendors to be used at different replicas.

- Optimistic execution of groups of operations. The idea is for read and write operations of a transaction to be executed optimistically in a single replica, then the results need to be validated at commit time. For that, the authors developed two algorithms that differ in how the master replica is chosen and how the operations execute optimistically in non-master replicas.

- Striping with BFT replication. This allows to take advantage of BFT replication to improve the performance of the read operations. The idea of this feature is to optimize read operations by stripping reads from different clients to different subsets of replicas.

Byzantium uses a BFT state machine replication protocol that requires 3f+1 replicas where each replica is connected to an DBMS. Both the BFT replication protocol and the DBMS used are considered as black boxes and Byzantium is the middleware that uses the protocol to make the interaction between clients and databases.

In order for Byzantium to work, it has to make sure that all the replicas execute the transactions against the same database snapshot. For that, the authors take advantage of snapshot isolation semantics to ensure that all the operations inside each transaction will see a consistent state of the database. Besides that, when the BEGIN, COMMIT and ROLLBACK operations are executed, the state machine BFT replication protocol used enforces total-order on the messages corresponding to those operations.

Byzantium creators developed two algorithms, single-master and multi-master, where their main difference is on how the master replica is chosen and how operations are executed optimistically in non-master replicas. While the single-master approach is aimed to perform better for read-write dominated workloads, the multi-master version performs better when there is a large number of read-only transactions.

2.3.3.1 Multi-master optimistic execution

In the Multi-master optimistic execution, the master selection can be different for each client, leading to more flexibility since the masters can be picked in order to provide better load balancing. The way this algorithm works is as follows:

1. The master is selected in the beginning of the transaction, at random or according with a sophisticated load-balancing scheme.

2. Subsequent reads and writes are performed optimistically at the master replica.

3. When the transaction attempts to commit, two validations are performed:

(a) The client obtains the correct value. To ensure this, all non-master replicas execute each transaction operations and verify that the returned result match the results previously sent by the master. Due to the fact that both BEGIN and COMMIT operations are serialized by PBFT, if the master and the client are correct, then all correct replicas should also have obtained the correct result.

(b) the transaction can commit according to Snapshot Isolation. If the database has optimistic concurrency control, then guaranteeing Snapshot Isolation is immediate. However, most database system rely on locks for concurrency control. So, when a replica is acting as a master for some transaction, it will have to obtain a lock on the rows it changed. This situation, can block the local execution of a committing transaction that has written in the same rows, leading to a deadlock.

Byzantium solution for the deadlock problem is based on the savepoint mechanism which allows to rollback all operations executed after the savepoint was established. So the solution relies on temporarily undoing all operations of the ongoing transaction. After executing the committing transaction, if the commit succeeds, the ongoing transaction is rolled back due to a conflict. Otherwise, we replay the undone operations and the ongoing transaction execution may proceed.

2.3.3.2 Single-master optimistic execution

Unlike in the multi-master approach, in single-master all the transactions are optimistically executed in the same single node.

With this comes the problem of leader election, where clients and replicas must agree on a single master that should execute all the reads and writes during a transaction. The way Byzantium solved this was by augmenting the service state maintained by the PBFT service with this information. They basically mixed PBFT leader election mechanism with their protocol and augmented the service interface with special functions to allow them to change the current master.

The basic way of operation of the algorithm is as follows:

1. The master is elected according with the PBFT mechanism and is sent along with the BEGIN message.
2. The client sends the operations to all the replicas. It also stores them locally until the end of the transaction.
3. All read and write operations are executed only by the master replica, then it sends back the result.
4. The client will received and store the result until the end of the transaction.
5. When the client sends a commit message, it will send along with that message the hash of all the operations and results received from the master replica.
6. At commit time, all non-master replicas will execute all read/write operations of the given transaction and then compare the hashes of both operations and results with the ones received by the client. If they match then the transaction is committed successfully.

In this scheme, if a transaction commits in the master replica, then it will be able to commit in all non-master replicas independently of the concurrency control mechanism used. This is due to the fact that all the replicas are committing the transactions serially. If the replicas are working correctly, then they will all obtain the same results.

2.3.3.3 Dealing with a faulty master

In the presence of a faulty master, the results received by the client are unreliable since they can be erroneous or no results are received at all. To be assured that the results received by the master are faulty, at commit time each replica compares the master's hash of all results with the hash of its own results. If they do not match then ideally the client will receive a majority of messages notifying that the commit failed. In a situation like this, it is possible to distinguish the leader message from the remaining ones and identify it as malicious/Byzantine if that's the case.

Another problematic scenario is when the master does not reply to the message. However, we typically wait for all messages to be delivered to the client. If the client reaches the timeout period and the leader message has not been received yet, then a leader change should be proposed by the client (it should be proposed by the replicas). Note that any transaction that was in undergoing should be cancelled/redone.

2.3.3.4 Dealing with a faulty client

A very simple procedure to be done when dealing with possible faulty clients is to check for a valid sequence of operations from each client. For example, a BEGIN operation must always be followed by a COMMIT operation. (BEGIN -> COMMIT)

Another interesting scenario is when a faulty client sends different operations for each replica, at commit time, only the replicas with the matching hash of the set of operations will commit the transactions. This problem would lead to different states in each replica since the databases from each replica would not be all in the same state. The idea here is to allow the replicas to vote whether they have the same sequence of operations that match the digest sent by the client or not. If 2f+1 agree on the fact that they hold all the operations, the transaction will commit and correct replicas that were not on the set that agreed must obtain the set of operations from the other replicas. Also, if the primary checks that it does not have the correct sequence of operations, then the whole transaction must rollback on all replicas. If there is no agreement on the sequence of the operations, then a primary change must occur and the primary will repeat the whole process.

Another situation that can be problematic is if the client sends the incorrect hash of the results to the replicas which could lead to the master committing the transaction and the remaining replicas not. A simple way to fix this is for the primary to only commit if the hashes he received match the hashes he previously sent.

2.4 Final Remarks

After going all this contextualization, reviewing the related work approaches and studying each of them, we believe that the work developed by Byzantium authors is what resembles the system we planned on developing. It has very interesting characteristics:

- It does not rely on any trusted component like HRDB
- It proposes two very interesting approaches to handle transaction processing
- These algorithms take advantage of Snapshot Isolation semantics that allow these algorithms to have very good performances without violating the isolation property from ACID.

So, given the tools we have at our disposal, we are clearly inclined to build our Diverse Intrusion-Tolerant Database Replication system, based on one of Byzantium transaction handling algorithms (which we will discuss in the next chapter). We will also approach other problems related with the development of a replicated database system, but there is not much literature about them since they have not been explored in this current context.

Chapter 3

DivDB

In this chapter we will cover the design and implementation of DivDB - Diverse Intrusion-Tolerant Database Replication system. We will go through our architecture and explain all its features. We will explain how we implemented our JDBC driver and how it abstracts the user from most of the BFT protocol and mechanisms details. Besides that, we will also give a brief explanation of Byzantine fault-tolerant state machine replication library we chose to use.

Another important section of this chapter is the open problems that we chose to approach, by discussing them, presenting alternatives and explaining our design choices.

3.1 DivDB Overview

The basic idea of our system design is to develop a JDBC driver that communicates with several replicated databases, while abstracting whoever is communicating through it of all the complexity it involves. Each database can be from a different vendor, thus the diversity component of the project.

In order to design our system in the best possible way, we had to study and analyse several architectural aspects. First, we had to analyse witch kind of JDBC driver to build. The one that suited the most our system was the JDBC type-3 driver. As it was described in the previous chapter, the type-3 driver is composed by a middleware that is responsible for communicating with multiple DBMS. That is exactly want we want for our system.

The JDBC driver abstracts the application from all the BFT protocol and mechanisms details. For the user application it is as if they were communicating with an DBMS like any other.

An essential component in our architecture is BFT-SMaRt [31], which is a BFT State Machine replication library. We will discuss it in more detail later.

Figure 3.1 shows that our architecture is based on a 3-Tier model since basically it is split in three tiers:

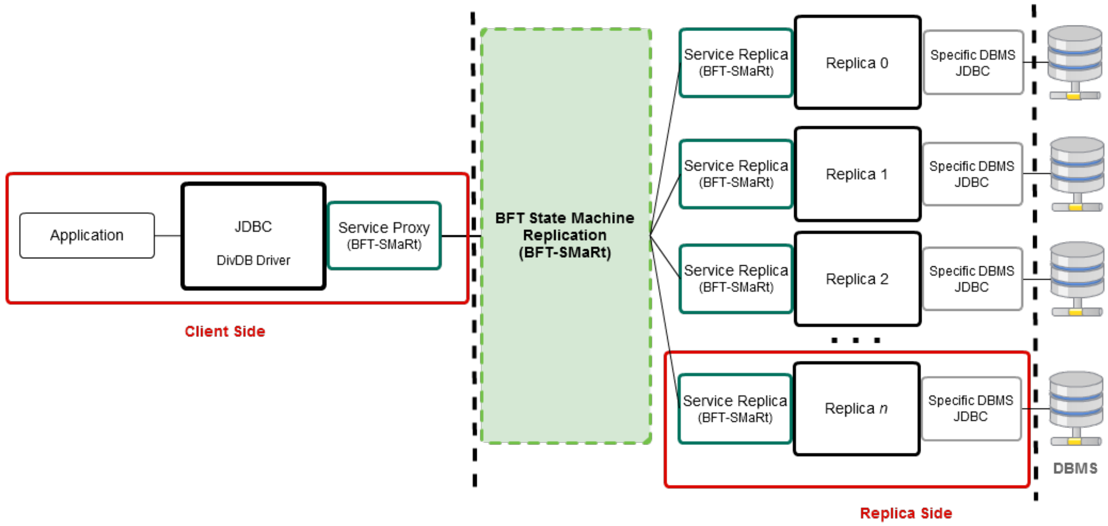

Figure 3.1: DivDB Architecture Overview

- **Client Side:** The client machine hosts the application that will communicate with the DBMS through the JDBC driver.
- **Replica Side:** Each replica machine receives the requests, forwarding them to the DBMS and sending back the reply to the client side.
- **DBMS:** The actual database, this is where all the requests are executed.

It is important to note that there are some requirements that the DBMS must fulfil in order to be used by our system. We want to achieve diversity, but the DBMS to be used must be be compatible in certain aspects. They must all possess a JDBC driver, since it is the interface we will be using to communicated with the database. They must either be compatible with MySQL syntax or their commands must have a corresponding command in the MySQL syntax. Besides that all the databases to be used must support Snapshot Isolation semantics. Any database that fulfils these requirements can be used in of the DivDB replicas.

3.2 Open Problem

The development of an Intrusion Tolerance Database Replication System has many challenges. Given the complexity to develop such system, we decided to select some of them we though relevant and interesting.

This chapter will be focused on the discussion of actual problems of designing an Intrusion Tolerant database replication system such as the authentication procedure, transaction handling and state transfer. We will explain the difficulties we were confronted to when trying to solve each of these problems and how we attempted to solve them.

Later on, all this research will be used in the actual implementation of these features in the development of our own JDBC driver which masks from the user any IT mechanism. In other words, for the user it will be as if he is using a JDBC driver like any other. These architectural implementation details will be further discussed in Chapter 4.

3.2.1 Login and authentication procedure

Login and authentication is an important feature in DBMS because it allows us to know that the user accessing the Databases is legitimate. Besides that, it enables one do access control by limiting the privileges and powers that each user has inside the DBMS.

At a first look, the login procedure to a DBMS seems to be a very simple procedure. All an user has to do is specify the username, password and database name. In our specific case, we want to connect through JDBC interface, we have to specify the username, password, database URL and the JDBC driver string.

However, we want a client to be able to connect to our Diverse Intrusion-Tolerant Database Replication system. That is where things start to get complicated because each replica is connected to its database and each database has its own username and password, in order to ensure fault independence. So, the main challenges we found when studying how login procedure would be done were:

- **Connect to the databases in each replica**: The idea here is to be able to connect to each replica and minimize the amount of work to be done. In order to do this, we chose to use the existing fields and separate the login information for each replica with a semi-colon (;):

  Username: *"<Replica-1 Username>;<Replica-2 Username>;...; <Replica-n Username>*

  Password: *"<Replica-1 Password>;<Replica-2 Password>;...; <Replica-n Password>"*

  Database URL: *"jdbc:divdb;<Replica-1 URL>;<Replica-2 URL>;...; <Replica-n URL>"*

- **Isolated Login**: When a client sends the login and password, due to the BFT protocol scheme, every replica is be able to see each other authentication data. So if a replica gets compromised, all other replicas can also get easily compromised since an attacker could easily obtain all other replicas login information. To overcome this we use a shared key (symmetric cryptography) between each client and replica in order to hide each pair of user/password from other replicas. In other words, each pair of user/password is encrypted with the proper shared key using the AES-256 encryption algorithm and each replica can only decrypt the user/password destined to itself.

- **Recovery from unsuccessful login:** When a user inserts the authentication information to be used to login into the replicas databases, it is important to try to ensure that the login was successful in all databases. There could be a human mistake in some of the logins or even a problem in one of the replicas and the system might still work. This is because our intrusion-tolerant system is designed to tolerate f faults in a total of 3f+1 replicas. However, it would make the attacker's task of compromising the whole system much easier (less replicas to compromise). The system was designed so that this fault tolerance feature is only enforced after the login procedure. Before that the system is designed to have zero fault tolerance.

This way, all replicas must do login procedure successfully. If this does not happen then the client is notified about the problem.

- **Session Manager:** The main idea is to develop a component that keeps track of each user actions and tunnels each connection to the databases. The task of this Session Manager is to have a table to keep track of each clients connection to the database. It also stores all operations until a COMMIT is sent, if that is the case. This way, it associates each client action to a connection and therefore mediate the access to the database. This table also allow us to do proper logging of each client actions which could be used later to detect any malicious actions by a client.

As we can see, the deeper we explore these problems the more complicated they get. Each solution we come up with, other problems may arise with it. One particular problem that came up in one of our solutions was the shared key. How each pair of client/replica would know their shared key. There could be ways to establish a shared key without the other replicas knowing, since the BFT-SMaRt Library already uses cryptography mechanisms to sign their messages. However, it did not offer much interfaces to access them so we simply assumed the keys to be pre-shared.

3.2.2 Concurrent transaction handling

Conventional DBMS already implement one ore more transaction handling mechanisms which can grant different degrees of transactions isolation. However, we can not simply rely on these mechanisms when we are talking about a replicated IT system. After studying several possibilities, we came up with an interesting alternative: to convert transactions to stored procedures and add them to each database.

3.2.2.1 Transactions vs Stored Procedures

We studied the pros and cons of using stored procedures in order to know if we could benefit using stored procedures instead of transactions. The benefits that one can take advantage of when using stored procedures [7, 12] are:

- Reduction of the network usage between clients and database servers. A slow network connection can undesirably affect throughput of a database. Stored procedures help minimize the amount of data interchanged between the client and the server because all of the code to be executed is already on the database side. So, while one only has to do one request to call a stored procedure, a raw SQL may require several distinct interactions with the DBMS server to achieve the same request. Therefore, by encapsulating the set of operations within a stored procedure, regardless of where the invoker is located on the network, the impact on the network traffic is minimized.

- Simplification of blocks and deadlocks since it makes it significantly easier to keep deadlocks under control if an entire database transaction is performed within a single database request.

- The addition of an extra layer of abstraction into the design of the databases. This extra layer can hide some details regarding the database from malicious users such as definitions of tables. In other words, they can avoid the exposure of database content if every communication to the database is done through stored procedures.

On the other hand, some of the disadvantages are:

- Stored procedures of most vendors do not really follow the standard. In other words, the language/syntax of stored procedures varies greatly from vendor to vendor. In practice, switching to use another vendor's database or using multiple databases from different vendors would most likely require rewriting the stored procedures in several different languages.
- Stored procedure languages from different vendors may also have different levels of sophistication. For example, stored procedures from a certain vendor database might have more language functionalities and features than some other vendor's database.
- They go against the idea of a three-tier architecture. In the three-tier architecture there is the presentation, logic and data tier, Typically all the logistics should be done at the logic tier, but by using stored procedures we are moving part of the logic tier to the data tier. This overloads the data tier with logical computations whereas it should only overload it with storage-related computations.

After taking into account both advantages and disadvantages, we chose not to adopt the stored procedures mainly because one of our goals is to have diversity. So, the lack of standardization in stored procedure syntax and language complicates the development of a database and the stored procedures to interact with it since if we have *n* different replicas, each with a different DBMS, it will be very likely that we will have to develop *n* different versions of the stored procedures, one for each different DBMS.

3.2.2.2 Transaction handling

The problem of transactions in the context of replicated intrusion tolerant databases is concurrency because we have the same operation being executed at several replicas. Having concurrent transactions being executed over several replicas might lead to incoherent results among each replicated database. There are previous proposals that do allow transactions, they either are not capable of executing transactions concurrently or they rely on a trusted coordinator node that handles the requests and forwards them concurrently. Not executing the transactions concurrently limits the performance of the overall system since the transactions must be executed serially. This approach was proposed at [32].

The most viable ways of solving this concurrency problem are:

1. Using a **trusted coordinator** like we have previously mentioned. In [3] they provide a BFT database replication prototype that uses a trusted entity (Shepherd) which includes both the replica managers and the coordinator. Basically, the coordinator handles the requests and

forwards them concurrently to the replica managers. This is done in such way to maximize the parallelism between concurrent operations which enables HRDB to achieve good levels of performance.

2. A **multi-master algorithm introduced** in [5]. In this approach, each client select a different replica as master which provides more flexibility in terms of load balancing since it will not be the same replica handling the requests from all the clients. However, most database systems rely on locks for concurrency control. So when a replica wants to change some rows, it must first obtain locks on those rows so that it can modify them.

3. The **single-master solution**, also presented in [5], executes optimistically the transactions in the same single node. This node is chosen by the replicas and clients and it will execute all the operations of the transactions first. Then at committing time, it executes on all replicas the transaction. The basic idea here is that if the transaction commits in the master replica it will also commit in all correct non-master replicas. Besides that, since all replicas will follow the same order of execution of transactions as the master, concurrency problems are avoided.

After analysing each approach We believe the single master approach since it is the most suitable solution in the context of our implementation.

Regarding the first approach, the fact both the coordinator and replica managers assumed to be trusted makes it an assumption hard to ensure. If the coordinator was not assumed to be trusted then it would be a single point of failure. So, if the coordinator fails, the whole system availability is affected until the coordinator is fixed. Besides that, the use of a coordinator node would make the architecture centralized around a single node whereas in our design we want it to be as distributed as possible.

As for the multi-master algorithm where we must extract the write-sets when we want to modify something. It relies on the use of locks to obtain write access to the rows it wants to modify. However, this approach can lead to a deadlock. So, in [5] they implemented two mechanisms to avoid deadlocking: one uses database triggers to extract write-sets and the other one is based on the analysis of the SQL code. However, this multi-master version had some issues with an unimplemented method in the JDBC interface when using HyperSQL as DBMS in the replicas. That JDBC driver did not support a required method to handle the write-set feature. So, they had to develop a workaround which led to a performance deterioration when running the TPC-C benchmark. Since we will be using diversity in our implementation, if we were to base our transaction handling in the multi-master way, we would either deteriorate the performance of our system or limit the use of certain DBMS which clearly goes against our goals.

So, given this limitations, we developed our transaction handling mechanism using on the single-master algorithm from [5]. The results from that research show that the performance from single-master is very similar to the multi-master approach. Which means that in terms of performance it is not big disadvantage to have chosen the single-master. Also, the single-master algorithm is easier to implement in our architecture.

By default, most DMBS come with auto commit mode on. The one way we can look at this is as if was single operation is a transaction. However, when executing a transaction auto commit feature has to be off (All commands after a BEGIN until the COMMIT operations are not automatically committed until after the COMMIT itself.).When doing transactions through JDBC interface, there is no command for the BEGIN operation. What happens is that all operations until a COMMIT is sent are part of a transaction. In other words, after each COMMIT there is a BEGIN implicit.

First of all, the replicas and the clients must agree on a master replica. This can either be done randomly or we can establish a picking order for the replicas according with each DBMS efficiency since that replica will be executing all incoming requests. So the faster it finishes it, the faster non-master replicas can know if they should execute or not the transaction.

Note that the ideal way of picking the master replica would be through a leader election scheme. Although this master replica has nothing to do with the BFT protocol primary replica, one easy way to conciliate them both is use them as the same, and call for BFT protocol leader change any time the master replica is detected to be faulty. However, since the BFT-SMaRt library does not have any interface to call leader change methods, we left this solution for future work.

We now present a more detailed description on how the algorithm works:

Algorithm 3.1 Transaction Processing Algorithm

1. The driver sends operations to the replicas. All these operations are stored in a Java table until COMMIT time.
2. The replicas receive the operations and proceed as follows:
   (a) if it is the master replica, execute the operation and send back the result. Both the operation and the result are stored in separated table.
   (b) if it is a non-master replica, store the operation in a table and send back a default value.
3. The driver will fetch the master replica result and send it to application level. It will also store in a table all the results received from the master replica.
4. At COMMIT time, the driver will send totally ordered COMMIT message along with the hashes of the two tables: operations and results.
5. When the replicas receive the COMMIT message:
   (a) if it is the master replica, it will calculate the hash of its operation and result tables and compare with the hashes it received from the driver. If they match then the transaction COMMITS.
   (b) if it is a non-master replica, it will calculate the hash of the operations table, compare with the one received and if they match it will execute all the operations. At the end, it will calculate the hash of all the results and compare with the results hash received from the driver. If they match then the transaction COMMITS.
6. If the driver does not receive at least f+1 confirmations it will issue a ROLLBACK command.

The main idea behind this algorithm is that each totally ordered message induces a new state into the state machine replication system. So if all replicas were in the same previous state, a COMMIT message will take them all into a new state which is equal for all of them since they all executed the same transactions in the same order.

3.2.3 State-Transfer

State-Transfer plays an important role in Intrusion-Tolerance because if a system merely tolerates faults, it will eventually become unreliable after tolerating a certain amount of faults. This is where recovery mechanisms come in. They enable the system to recover faulty replicas and maintain its proper execution state.

The approach we followed and implemented is very simple. However, it is also very heavy in terms of processing and I/O. Basically it requires to extract all the data from the database. in other words, we have to dump all the contents from one database and load them into another. The way it was implemented is described bellow in Algorithm 3.2.

Algorithm 3.2 Full data transfer solution

1. Whenever state-transfer is called, one of the replicas will extract all the information from every table of the given database. In order to dynamically identify all the tables from the database, we access the database metadata to discover the tables and their columns. Then all the data is extracted through SELECT statements.
2. All the data obtained is then sent to the replica to be recovered.
3. The replica that is going to be recovered must first clear out all the table from its database. Once this is done, INSERT VALUE statements are used to put all the data into the database.

Each SELECT statement returns a `ResultSet` and since we chose to develop our `ResultSet` implementation (see Section 3.3.2) to cache all the results from each query into memory (instead of keep accessing the DBMS), this solution fitted us perfectly.

We also studied another possible approach in this matter to solve state-transfer. The idea would be taking advantage of a database feature called savepoints

A **savepoint** is a way of implementing sub-transactions (also known as nested transactions) within a DBMS by indicating a point within a transaction that can be "rolled back to" without affecting any work done in the transaction before the savepoint was created. The way this would work is as follows:

1. From time to time a save point is created and all the transactions executed inside that savepoint are stored.
2. One replica will send all the stored operations to the replica to be recovered.
3. The replica that is going to be recovered rollbacks to the latest savepoint and executes the set of operations.

Although this approach can be more complex, it is also more lightweight since for bigger databases, transferring the whole database contents over the network can be costly. However this is still an idea proposition and we leave it here for possible future work research.

3.3 DivDB Information Flow

In this section will cover in a logical way, how the information flows in our system. Like any other JDBC driver, the user typically interacts with it (e.g., sends queries, updates some rows, etc.) and the driver communicates with the database. In our system the procedure is similar. However, between the JDBC driver and the database lies our entire middleware.

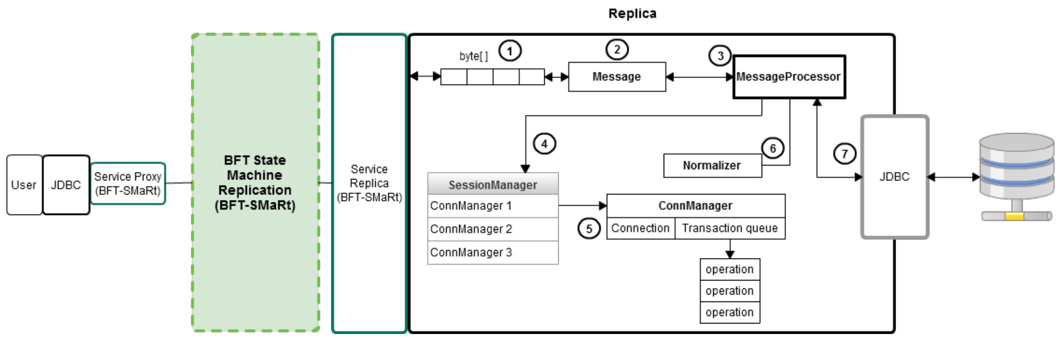

Figure 3.2: Work-flow inside replica

Figure 3.2 illustrates the flow inside each replica when it receives a request. Although some of the names in the Figure 3.2 are actual classes of our DivDB, here we are only presenting a logical flow of the data and information inside each replica. The steps taken when a replica receives a request are:

1. The replica receives an array of bytes through BFT-SMaRt interface.
2. The array of bytes is then transformed in a **Message** object. This object is understandable by the replica, unlike the array of bytes.
3. The message is then processed by the **MessageProcessor** according with the kind of request.
4. The **SessionManager** then fetches the **ConnManager** responsible for the connection to where the message is going to be sent.

After step 5 the execution steps may vary depending on the kind of message:

- Regular message:
  - Step 5: The message is stored in a queue of operations, where all the operations of a transaction are stored.
  - Step 6: The master replica will normalize the request. (non-master replicas only normalize at COMMIT operation)
  - Step 7: The master replica sends the SQL operation through JDBC driver to be executed at the database. (non-master replicas only execute at COMMIT operation).

- Commit message:
  - Step 5: All the messages from the queue are fetched.
  - Step 6: All the messages from the queue are normalized (master replica does not do nothing at this step)
  - Step 7: The non-master replicas execute the operations through the JDBC driver (the master replica does not do this). The hashes of the results and operations are compared with the ones received from the client. If they match then the commit is sent to the DBMS. Otherwise an exception is sent to the client.

This description shows how the information flows inside the replicas, during the execution of the Algorithm 3.1 present in the previous section. The purpose of this explanation is to give a high-level view of how things are supposed to work inside the replicas.

3.4 DivDB Implementation

In this section we will describe in more detail how we implemented DivDB, how it is structured internally by describing the Java classes that are part of it and their roles in the system.

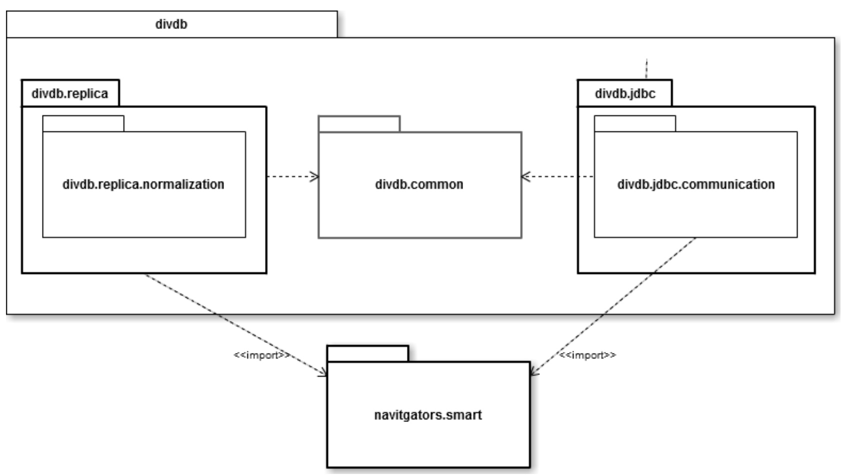

Figure 3.3: Java Package Diagram

The DivDB implementation was designed to be as modular as possible and with the least amount of dependencies between packages. The figure 3.3 depicts the Java packages organization and dependencies. First of all, the `divdb.jdbc` package basically contains all the implementations of the interfaces necessary to have a working JDBC driver type-3. As for `divdb.jdbc.communication`, it is responsible for bridging the JDBC-related classes with BFT-SMaRt client side interface through the package library `navigators.smart`. The `divdb.jdbc` package contains all the essential classes for the client side of the system.

The name of the `divdb.replica` package is self explanatory. It contains the classes essential for the replica side. Inside it there is a package `divdb.replica.normalization` which is responsible for uniformizing the SQL syntax of different DBMS. The `divdb.replica` package, like `divdb.jdbc.communication`, is dependable on the `navigators.smart` package library in order to be able to communicate through BFT-SMaRt.

The package `divdb.common` contains essential data-structures and classes needed both at client and replica side. So both `divdb.jdbc` and `divdb.replica` depend on it.

Packages	Lines of Code
`divdb.jdbc`	1924
`divdb.replica`	724
`divdb.common`	3069

Table 3.1: Packages lines of code distribution

The DivDB implementation is comprised of a total of 20 classes and interfaces. These classes are distributed mainly amount three packages, as one can see in Figure 3.2. The total amount of lines of code produced is 5717, excluding white spaces and comments to the code.

Although most of the application logistics is done at replica side, it is interesting to note that the `divdb.jdbc` package has more lines of code produced than `divdb.replica`. This is due to the fact that most of the classes in the first package must implement certain interfaces, which come coupled with a lot of methods that must be there even though they are not operational. As for the `divdb.common` its huge size is due to our implementation of `ResultSet` (it is in this package because it is used to transfer `ResultSet` objects over the network to the client side) which is pretty much as fully functional as any other `ResultSet` implementation from any other DBMS.

3.4.1 BFT-SMaRt library

The BFT-SMaRt is a replication library written in Java. It implements a state machine replication protocol based on a modular protocol similar to PBFT [31]. The BFT-SMaRt is designed to tolerate Byzantine faults, while still being highly efficient, even if some replicas are faulty. In other words, if we have **n** replicas, such that **n = 3f + 1**, then we are able to tolerate up to **f** faults.

In order to be able to use BFT-SMaRt library functionalities, one must implement it properly at both client and replica side. As Figure 3.1 shows:

- `ServiceProxy` is the interface the client side must implement in order to be able to interact with the replicas. This interface provides the methods that enable the the client side to invoke commands to the replicas (these commands can be totally ordered or not).
- `ServiceReplica` is the skeleton interface for each replica. It defines the crucial methods that will be invoked to execute the operations the client invoke. It is able to distinguish a totally ordered command from a normal and react differently to it.

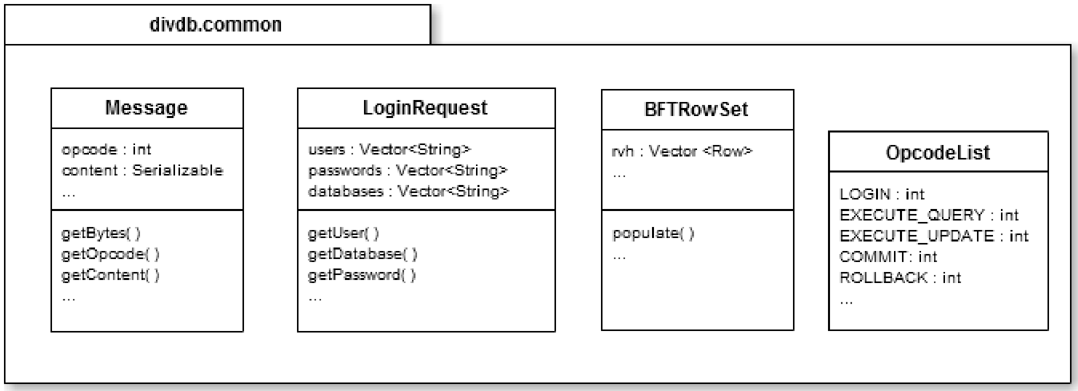

Figure 3.4: divdb.common class diagram

The BFT-SMaRt is in continuous development. This library was not only chosen because it its being developed by faculty colleagues and its current features, but also because of its portability. The fact it is written in Java couples very well with our diversity goal. Although we did not included in our research Operating Systems diversity [25], our system is very well capable of supporting it.

3.4.2 Common classes implementation

In this subsection we will present a set of classes that were used both at client and replica side. Therefore, these classes were grouped the package named `divdb.common` that can be seen in Figure 3.*4.* The classes that belong to this package are:

- `LoginRequest`: This class is an data structure to be used when a login message is sent. It properly stores the encrypted username, password and database name of each replica in a structured way.
- `Message`: This class is another data structure and the building block for communication of our system. It contains an Operation Code to identify what type of `Message` it is. The only restriction it makes about its content is that it has to be serializable, otherwise it could not be transferred over the network.
- `OpcodeList`: This class is basically a list of all the operation code that the system supports. It makes correspond a constant variable to the number that is the operation code.
- `BFTRowSet`: This class is an implementation of the Java interface `java.sql.ResultSet`. Every time we query the database, we obtain a `ResultSet` object back. Although in most implementations, the data is not actually in memory, it stays at the DBMS, in our implementation we fetch all the data and store it in memory. This is done so that the data from the `ResultSet` can be all transferred at once to the client-side and also because this way we can also use this `ResultSet` in our state-transfer mechanism.

3.4.3 Client-side implementation

The client-side of the system is mostly agglomerated inside the `divdb.jdbc` Java package. We will go through each class to explain their role. Figure 3.5 illustrates the class diagram of the client-side Java classes.

These classes, along with the ones in the `divdb.common`, can be stored in a jar file so that it can be used as library to connect to our system through a JDBC interface.

The classes belonging to the `divdb.jdbc` *package are:*

- `BFTDriver`: This class implements the interface `java.sql.Driver` and it basically loads up the driver configurations and connects to the database. This is where all the login cryptography mechanisms are applied.
- `BFTConnection`: This class implements the interface `java.sql.Connection`. It is the representation in Java of a connection to the DBMS. Part of the transaction handling process is done in this class, at the `commit` method.
- `BFTStatement`: This class implements the the interface `java.sql.Statement`. Our implementation simply creates the messages with the proper operation code and forwards them to the `MessageHandler` class.
- `BFTPreparedStatement`: This class is pretty similar to the previous one. It implements the interface `java.sql.PreparedStatement`. An interesting design choice in this class is the fact that all the variable attributions in the statement are done locally. This way we avoid exchanging a lot of message just to build the statement. Another important detail is regarding batching, to also avoid a lot of messages being sent, all the batch statements are stored and processed locally. Only when the `executeBatch()` method is called, all the batched statements are sent and executed at the replica side.
- `BFTDBMetaData`: This class was developed mainly because the application benchmark we used accessed it to retried information about the DBMS. It is an implementation of the Java interface `java.sql.DatabaseMetaData`. The implemented methods are all called locally without the need of contacting the replicas (e.g.,`getDatabaseProductName()`).

Inside the `divdb.jdbc` package there is another package called `divdb.jdbc.communication`, which handles all the communication with the replicas through the `ServiceProxy` interface. The classes from this package are:

- `MessageHandler`: This class plays a very important role in the client side. It works as a bridge between BFT-SMaRt Library and any class that wants to communicate with the replicas. BFT-SMaRt Library only allows arrays of bytes to be transferred through the network. However, the `MessageHandler` is responsible for abstracting the other classes from sending byte arrays, instead `Message` objects are sent to `MessageHandler` so it can convert them and send through the BFT-SMaRt.

- `BFTComparator`: This class role is quite simple and at the same time important. This class only has one method, `compare()` which is responsible for comparing the replies of each replica, and if there are enough identical replies, the `BFTExtractor.extractResponse()` is called.

- `BFTExtractor`: The role of this class, which only has one method too (`extractResponse()`), is to extract one reply out of the set of identical Messages received. Although this might seen redundant, this class is used in combination `BFTComparator`. Which enables it to extract specific replies with certain extra information that was not used in the comparison.

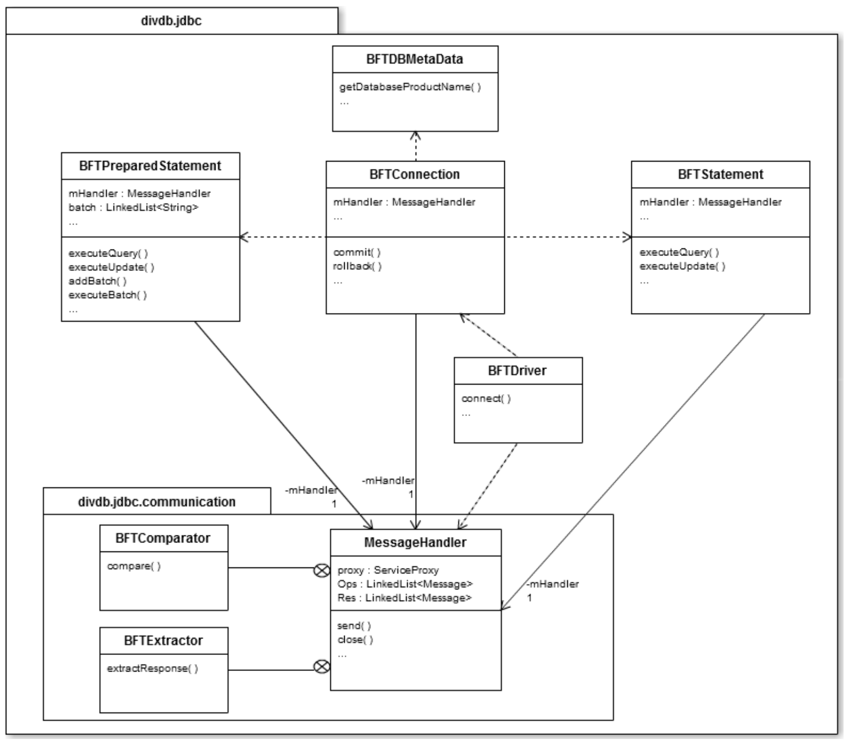

Figure 3.5: Client-side class diagram

3.4.4 Replica-side implementation

The replica-side is packed up in one single package. As can be seen in Figure 3.6, the package that contains all replica related classes is called `divdb.replica`. Besides that it also contains another package inside it called `divdb.replica.normalization`. The classes from `divdb.replica` package are:

- `Replica`: This is the main class of the replica-side, it implements the `ServiceProxy`, so it offers the methods to execute both totally ordered (`executeOrdered()`) and unordered operations (`executeUnordered()`). It is responsible for identifying which kind of operation is to be executed and forward it to the appropriated method at the `MessageProcessor`. It also contains the primitives for the state-transfer mechanism, `setState()` and `getState()`.

- `MessageProcessor`: As the name of the class indicates, it is here that all the messages are processed. It has methods for each kind of operation and it is also here where most of the transaction handling mechanism is implemented, most of the algorithm code is in `processCommit()` method.

- `SessionManager`: This class, along with `ConnManager`, has the task of keeping track of each connection to the replica DBMS, using a `HashMap` data-structure. Besides that, if a given connection is operating in a no-auto-commit mode, then it will store all the operations until a commit is issued.

- `ConnManager`: This class is a building block of `SessionManager`. They were split for a matter of code organization. This class contains the database `Connection` object and a `Queue<Message>` of the operations for that connection transaction.

Regarding the `divdb.replica.normalization`, it contains a `Normalizer` interface. This interface defines a method called normalize which is used to convert operations names that may have different names in different DBMS. This is essentially to uniformize the operations to be executed accordingly to each DBMS. By default, MySQL syntax is the one to be used in our system. For example, `FirebirdNormalizer` is a class that converts different commands from MySQL syntax into Firebird syntax. This is an essential feature in a diverse system, however we do not not cover all the difference. this differences are to be approached and normalized in a per need basis.

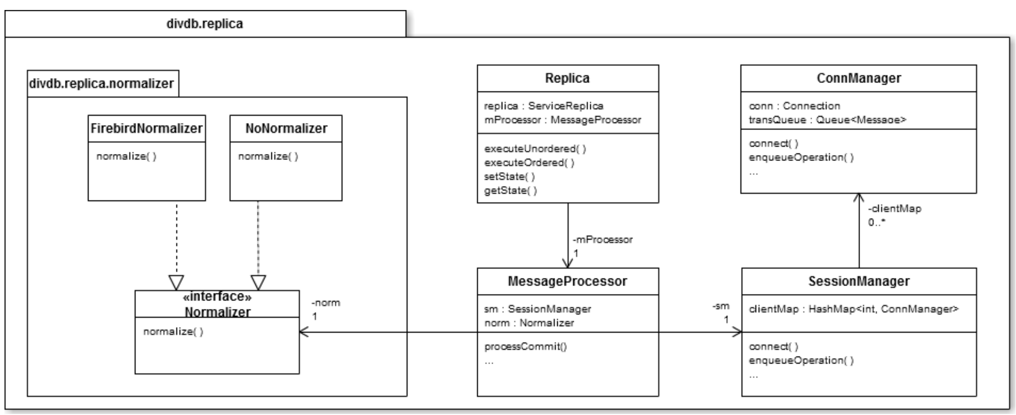

Figure 3.6: Replica-side class Diagram

3.5 Final Remarks

After going through our architecture overview, discussing the open problems related with the development of a replicated database system and explaining our architecture in detail, we can gladly say that we fulfilled our implementation goals. We managed to implement a robust and operational type-3JDBC driver. We carefully presented our approaches to the problems we decided to solve, and even discussed other possibilities to be considered in the future (e.g., using savepoints for state-transfer).

As for difficulties, we had 3 ambitious problems to solve, unfortunately the transaction handling mechanism was harder than we though to implement. Plus the fact that its implementation was not straightforward with BFT-SMaRt current interfaces, the development time of that phase was prolonged.

Regarding Byzantium mechanism to deal with faulty clients when they send different operations to each replica. In our implementation the replicas can detect at commit time that they do not match the ones from the master replica. However, their state will diverge from the correct ones. We did not implement this since BFT-SMaRt did not offer an interface for replicas to directly communicate with each other to agree on the operations. As for dealing with faulty master, it is able to detect it, but since we left leader change for future work, the system will stall if a faulty master is detected.

Chapter 4

Evaluation

In this chapter we will present the evaluation that has been done to the system that we designed and implemented in the previous chapter. For that we chose a widely known DBMS benchmarking tool, the TPC-C [33].

4.1 TPC-C

In this subsection we will briefly explain the TPC Benchmark C (TPC-C) specification [4]. The TPC-C is an Online Transaction Processing (OLTP) workload. The term OLTP refers to a class of systems that facilitate and manage transaction-oriented applications, typically for data entry and retrieval transaction processing.

TPC-C contains a mixture of both read-only and update intensive transactions that simulate the activities found in complex OLTP application environments. This is done by exercising several components that are used in typical from these kind of environments, which are characterized by:

- Concurrent execution of different kinds of transaction each with different levels of complexity.
- Online and deferred transaction execution modes.
- Multiple on-line terminal sessions.
- Moderate system and application execution time.
- Significant disk input/output.
- Transaction integrity (ACID properties).
- Non-uniform distribution of data access through primary and secondary keys.
- Databases consisting of many tables with a wide variety of sizes, attributes, and relationships.
- Contention on data access and update.

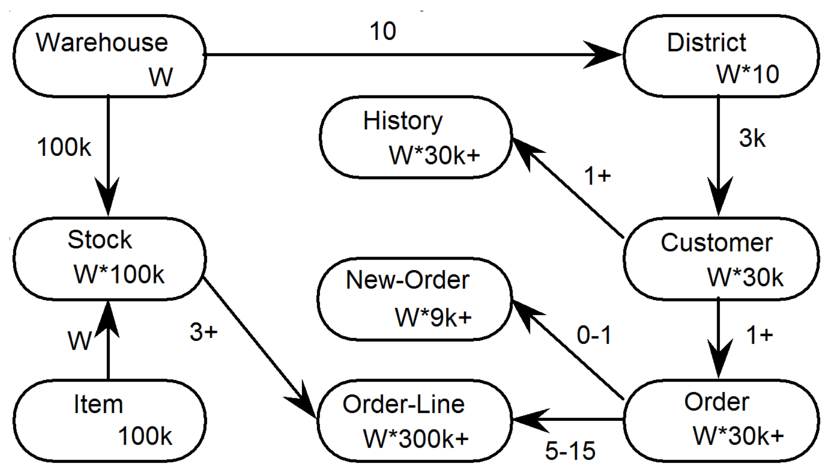

Figure 4.1: TPC-C Database Entity-Relationship diagram [4]

The components of the TPC-C database are defined to consist of nine separate and individual tables. The relationships among these tables are defined in the entity-relationship diagram shown at Figure 4.1.

The numbers present in the entity blocks represent the number of rows each table will have. These numbers are scaled by W, the number of Warehouses, to illustrate the database scaling. As for the number of W it will scale along with the number of terminals used to connect the the database. More specifically, for each active warehouse in the database, the benchmark tool must accept requests for transactions from a population of 10 terminals.

This benchmark contains five different transactions that operate over the database. Each transactions has different characteristics implying different levels of performance impact on the database. The TPC-C transactions are the following:

- **New-Order Transaction:** The New-Order transaction consists of entering a complete order through a single database transaction. It represents a mid-weight, read-write transaction with a high frequency of execution and strict response time requirements to satisfy on-line users. This transaction is the backbone of the workload. It is designed to place a variable load on the system to reflect on-line database activity as typically found in production environments.

- **Payment Transaction:** The Payment transaction updates the customer's balance and reflects the payment on the district and warehouse sales statistics. It represents a light-weight, read-write transaction with a high frequency of execution and strict response time requirements to satisfy on-line users.

- **Order-Status Transaction:** The Order-Status transaction queries the status of a customer's last order. It represents a mid-weight read-only database transaction with a low frequency of execution and response time requirement to satisfy on-line users.

- **Delivery Transaction:** The Delivery transaction consists of processing a batch of 10 new orders, that have not been delivered yet. Each order is delivered in full within the scope of a read-write database transaction. It has a low frequency of execution and must complete within a relaxed response time requirement. The Delivery transaction is intended to be executed in deferred mode, unlike the other transactions.

- **Stock-Level Transaction:** The Stock-Level transaction determines the number of recently sold items that have a stock level below a specified threshold. It represents a heavy read-only database transaction with a low frequency of execution, a relaxed response time requirement, and relaxed consistency requirements.

The amount of times that each of these transactions executes varies with the weight values attributed to them at the benchmarking tool. By assigning different weights percentages to each transaction we get different setup configurations which may allow to measure.

The performance metric for this benchmark is expressed in transactions-per-minute-C (tpmC) which represents how many New-Order transactions per minute a system generates while the system is executing four other transactions types (Payment, Order-Status, Delivery, Stock-Level). However, since most of research literature (e.g., [5, 3]) only measures the average of transactions per minute, we will also follow that approach. The tpmC metric is more conventional at business and corporation level.

4.2 Test Environment

In order to evaluate the performance of our system, our tests were done with 4 replicas, accordingly with the formula n = 3f+1, which would allow us to tolerate 1 faulty replica (f=1).

Since diversity was one of our goals, we also had to pick several databases in order to be used in our systems. The DBMS to be used were required to have a JDBC driver and be able to adopt Snapshot Isolation semantics. Our selections were also based on known studies and tests done using those databases [3, 5, 6]. All the databases we chosen are actually open-source and bellow you can read a brief description of each one of them.

- **MySQL** [34] is a DBMS that runs as a server providing multi-user access to a number of databases. It is mainly written in C and C++. It is supported by several platforms and one of its main features is the fact it possesses multiple storage engines, each with different characteristics and advantages.

- **PostgreSQL** [35] is a DBMS pretty much alike MySQL, written in C. It is also has cross-platform support. One characteristic of PostgreSQL is that among all the databases we selected, it that implements the majority SQL2008 standards [36].

- **HSQLDB or HyperSQL**(Hyper Structured Query Language Database) [37] is a DBMS entirely written in Java. It obviously is cross-platform. So, it basically offers a small, fast multithreaded and transactional database engine with in-memory and disk-based tables. Like PostgreSQL, it also implements a large subset of the SQL2008 standard[38].

- **FirebirdSQL** [39] is a cross-platform DBMS written in C++. It essentially has 3 different architectures. However, the one we chose to use is called Firebird SuperClassic. We chose it because it operates as a single server for all client connections, multi-threaded with separate caches for each connection.

The tests were run on a cluster of machines, each one with a Intel Xeon E5520 2.27 GHz processor, 32 GB of memory, a 146 GB SCSI disk and a Gigabit Ethernet network interface . The machines were running the Ubuntu 10.04 Server 64-bit operating system, with the kernel version 2.6.32. The Java Virtual Machine we used was OpenJDK Runtime Environment version 1.6.0_18. The versions of the databases used were MySQL 5.1, Postgres 9.1, HyperSQL 2.2.8 and Firebird SuperClassic 2.5.1.

In our evaluation we used an open-source implementation of TPC-C[40] which was also used in [5]. In order the be in accordance with Byzantium [5] test environment, we made slight modifications to the benchmark:

- Added a 1 minute warm-up phase before starting the performance measurements
- Allowed clients to execute on different machines (maximum of 30 clients per machine).
- Added a inter-transaction arrival time of 200 ms. In other words, we added an interval of 200 ms between the execution of transactions in each connection.

Besides that, in order for the benchmark client to work properly in our cluster we had do some more complicated modifications to the source-code. Since the benchmark client application was GUI (Graphical User Interface) based, it was limiting our tests execution on the cluster. In order to fix that issue, we had to develop our own console interface for the benchmark. Note that all the logic part of the application remained untouched.

Besides that, our benchmark database configuration included 10 warehouses. All our experiments were executed 5 times, where the results presented are all the average of those runs.

As for the benchmark workload configuration, we based it in a mixed workload experimented by Byzantium [5], which is composed as follows:

- 50% read transactions: with 25% for both Stock-Level and Order-Status.
- 50% write transactions: with 27% for New-Order, 21% for Payment and 2% for Delivery.

4.3 Results

We will now proceed to the analysis of the results from our tests. The main goals of our experiments were to evaluate the performance of our JDBC driver when compared to regular non replicated JDBC drivers. We plan on analysing the overhead of just providing BFT replication and providing diversity along with BFT replication.

In order to be able to compare our system and measure how much overhead it induced, we first had to run tests directly to the databases. For that, we connected our benchmark client directly into

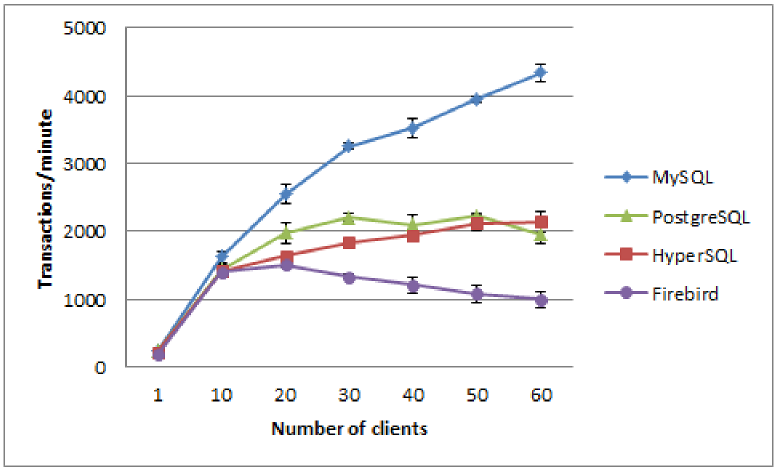

Figure 4.2: Performance on mixed workload. The benchmark client is directly connected to the databases through their specific JDBC drivers.

the databases and ran the tests. The results of those tests are illustrated in Figure 4.2. As you can see, MySQL has the best performance results. However, as for PostgreSQL results, they were a bit unexpected. According to [5], its results should be even better than MySQL. We tried to tweak PostgreSQL configuration but with no success. In order to attempt to find the reason for these result we ran some tools to monitor the machine hosting the database and found out that the disk write accesses were extremely high. We decided then not to investigate deeper this issue since it was out of the context of our work.

As for HyperSQL and Firebird, we had no previous results to compare them to, but clearly Firebird had some disappointing results, losing performance after the 20 clients. Regarding HyperSQL, we also monitored the machine that was hosting the database and the on the number of clients also noticed that with the increase of clients, the CPU usage and the amount of memory being used gets very high.

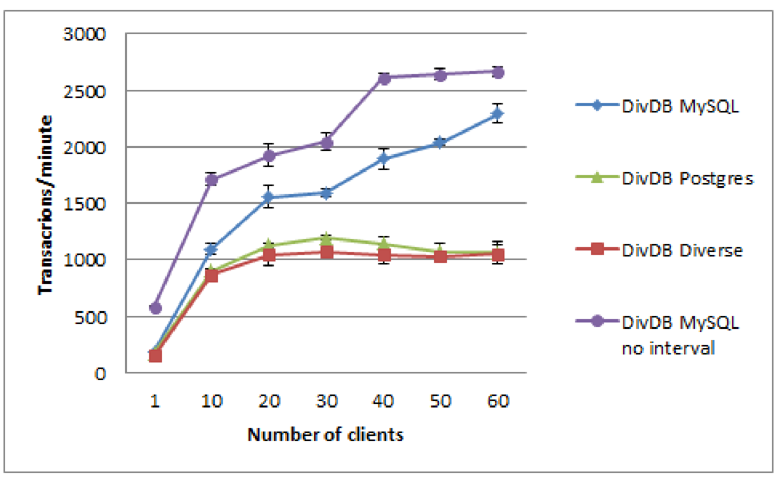

Figure 4.3: Performance on mixed workload. The benchmark client is connected to our DivDB system through the JDBC driver we developed.

The next step of our tests was to run the TPC-C benchmark client on our system. For that purpose, we did 4 different configurations presented in Figure 4.3:

- DivDB MySQL: In this configuration we used the MySQL database in all 4 replicas.
- DivDB Postgres: As for this one, PostgreSQl database was used in all 4 replicas.
- DivDB Diverse: This is our diverse solution using one different DBMS in each replica (MySQL, PostgreSQL, HSQLDB and Firebird)
- DivDB MySQL no interval: This configuration is the same as the first one, but we removed the 200 ms interval between transactions in each connection.

As we can compare from Figure 4.2 with Figure 4.3, both MySQL and PostgreSQL when compared with DivDB MySQL and DivDB Postgres respectively, they both have coherent results. After the 10 client mark, we can clearly notice that DivDB system introduces a 50% overhead. This can be seen as a good result. If we analyse carefully our transaction handling algorithm, the transaction is first executed at the master replica, and then only at commit time it is executed at the non master replicas. Therefore it is expected for our system to be around twice slower than the original DBMS.

Regarding DivDB Postgres results, they are worse than the ones obtained by Byzantium system [5]. However, our PosgreSQL results from Figure 4.2 were also worse. So we are simply using our own values as reference and they are clearly coherent with each other.

Now lets analyse the results from DivDB Diverse. The expected is that the performance of the system is crippled by the second worse replica. Since we are using 4 replicas, we are able to tolerate 1 faulty replica. In this specific case, the system simply does not has to wait for the replies of the slowest replica. Therefore, the system will not have to wait for reply from the slowest DBMS (Firebird in our case). So as expected, DivDB Diverse performance is around 50% slower than HyperSQL/PostgreSQL in Figure 4.2, since both of them got very similar results.

For a matter of curiosity, we also ran some tests to see the influence of the 200 ms interval. So, comparing DivDB MySQL with DivDB MySQL no interval. By comparing both results from Figure 4.3, we can analyse that without that inter-transaction interval, the system has an 20%-30% better performance. Showing that 60 clients is not enough to saturate a system based on MySQL we would like to remark that ,as mentioned before, the main reason we use it in our test is to be in conformity with [5, 3].

4.4 Final Remarks

The testing phase of this working was very time consuming. Small bugs arose when testing with a lot of clients and it took some time to fix them. Besides that, the tests done took us alot of time to complete. Every time we do a different test setup, we also have to load the databases with all the data. In our specific tests, a database with 10 warehouses has a size of almost 1GB, generating that amount of information and inserting it in a database is very time-consuming (more than 1 hour just to generate the database data for each test).

Regarding PostgreSQL, HSQLDB and Firebird, it would be also interesting to investigate and attempt to tweak them more in order to obtain better results with them. Speaking of results, it would also have been interesting to a more variety of results, play around with benchmark weights and draw out new conclusions.

Another set-back that we had was the benchmark client. As we have mentioned, we had to adapt the existing client that used a GUI interface in order to be able to run it in the clusters console terminal. For that we cleaned all the from code all GUI related code and wrote it to interact with the console instead. Previously the benchmark had 2604 lines of code and after our remodulation it went down to 2081 lines of code.

Chapter 5

Conclusion

In this thesis, we presented DivDB, a Diverse Intrusion-Tolerant Database Replication system, which combines State Machine Replication, BFT Transaction processing, and the use of different databases at each replica. We formulated, solved and implemented three problems: authentication, transaction handling and state-transfer.

Although transaction handling already had solutions proposed [3, 5], the fact is that no other BFT system solved all the 3 problems we approached and implemented them in a functional system like we did. Regarding login authentication, our implementation covers all the main issues except for key distribution. In the state-transfer problem, we developed a solution that works, however we gave an possible idea that could be used in the future to attempt to further improve our state-transfer mechanism.

Our current system is quite robust and the JDBC driver abstraction is almost fully functionally like any other from a regular database, unlike these other BFT system implementation where they do not even discuss having a JDBC driver implementation. Our JDBC driver can be used in simple OLTP systems, where they will benefit from an increased level of security, thanks to all its IT mechanisms, with an average downside of 50% decrease in performance.

It is also interesting to mention that our system is already being used by some colleagues with a library called Hibernate [41]. Hibernate is a library that does the mapping from Java classes to database tables (and from Java data types to SQL data types). It also facilitates data querying and retrieval. Hibernate generates the SQL calls and attempts to relieve the developer from manual result set handling and object conversion and keep the application portable to all supported SQL databases with little performance overhead. It operates on top of our JDBC driver and so far it has worked without problems.

The experiments we did with our system came to show us that it introduces an overhead of about 50% in the number of transactions processed per minute. Another interesting aspect is the fact that with diversity, even though some databases are slower than other, in the best case scenario, the system does not have to wait for the replica with the slowest database. So, the system performance is bounded by the second slowest DBMS being used.

5.1 Overcomed Dificulties

From the research phase to the development phase we had several difficulties that we had to overcome. Some of them were unexpected which delayed even further our work. Initially, we tried to use stored procedures as alternative to transactions. We even found some open source implementations of the benchmark using stored procedures. However, translating them to all the four different syntaxes each DBMS was too complicated and it did not couple well with the diversity goal.

Another difficulty we had was implementing the login procedure, where it would have to wait for all the replica. BFT-SMaRt did not offer ways of doing this, so we had to wait for it to be developed. This was solved with the introduction of the interfaces of the `Extractor` and `Comparator`.

One of our main difficulties was the implementation of the transaction handling algorithm. It was hard to debug and some problems only appeared with the increase in the number of connections. Due to the complexity of the system, this could only be tested in the cluster, which eventually we got access to. Then when working on the cluster, another problem arose, the benchmark client was GUI based and did not offer a console interface. This forced us modify the benchmark client code in order to operate on a terminal environment.

We still had some more tiny problems (e.g., for a lot of clients, some connections wait for a long time and call BFT leader change procedure) but eventually they were all dealt with.

5.2 Future Work

Although the development of the basic DivDB took more time than expected, there is still a lot of room for improvement. BFT database replication is a relatively recent area of research, depending on the IT mechanisms applied, new problems can arise. We established our system design goals, then studied the some of the possible problems that it could happen and we developed a solution for each of one. However, many of them still need to be solved, other appear as we research further on. So, here we will enumerate several points in the area of database replication and related with our work that could be further investigated and studied:

- Regarding the authentication procedure, a new problem came up, the establishment of shared keys. It would be nice to attempt either to incorporate this with BFT-SMaRt or use a sort of secret sharing mechanism.
- As for the transaction handling mechanism, study the possibility of executing optimistically some operations of the transactions at the non-master replicas. This could be applied to read-only transactions, in order to greatly improve their performance.
- Another problem from the transaction handling mechanism that was left unfinished was the leader election. Our suggestion for this is to wait for BFT-SMaRt developers to implement public primitives that allow one to invoke leader change methods. Then use the same leader the transaction algorithm as BFT-SMaRt uses internally.

- In the state-transfer problem, our solution is not exactly the most optimal, we would suggest exploring alternatives to find out if they are worthy, namely the savepoint approach we mentioned.

- Our current implementation of the `ResultSet` grabs all the results from the query and sends them to the client. It would be nice to implement a version of the `ResultSet` where it would send a message for every method invoked over it at client side, and then compare the performance to see if it is worth it.

Bibliography

[1] P. E. Veríssimo, N. F. Neves, and M. P. Correia. Intrusion-tolerant architectures: Concepts and design. In R. Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, volume 2677. 2003. (document), 2.2.1, 2.7

[2] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association. ISBN 1-880446-39-1. (document), 1.2, 2.2.2.2, 2.8, 2.9

[3] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. *SIGOPS Oper. Syst. Rev.*, 41(6):59–72, October 2007. ISSN 0163-5980. (document), 1.2, 2.3.2, 2.10, 1, 4.1, 4.2, 4.3, 5

[4] Tansaction Processing Performance Council. TPC-C current specification, April 2012. URL `http://www.tpc.org/tpcc/spec/tpcc_current.pdf`. (document), 4.1, 4.1

[5] Rui Garcia, Rodrigo Rodrigues, and Nuno Preguiça. Efficient middleware for Byzantine fault tolerant database replication. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 107–122, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0634-8. 1.2, 2.1.5.1, 2.3.3, 2, 3, 3.2.2.2, 4.1, 4.2, 4.3, 4.3, 5

[6] Ilir Gashi, Peter Popov, and Lorenzo Strigini. Fault tolerance via diversity for off-the-shelf products: A study with SQL database servers. *IEEE Trans. Dependable Secur. Comput.*, 4(4): 280–294, October 2007. ISSN 1545-5971. 1.2, 4.2

[7] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008. ISBN 9780131873254. 2.1, 2.1.1, 2.1.2.1, 2.1.3, 2.1.4.1, 3.2.2.1

[8] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995. ISBN 0-201-53771-0. 2.1

[9] Kevin Kenan. *Cryptography in the Database: The Last Line of Defense*. Addison-Wesley Professional, 2005. ISBN 0321320735. 2.1, 2.1.1

[10] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6): 377–387, June 1970. ISSN 0001-0782. 2.1.1

[11] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10715-5. 2.1.2.1, 2.1.4.1

[12] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing (Second Edition)*. Morgan Kaufmann Publishing, 2006. 2.1.2.1, 2.1.3, 3.2.2.1

[13] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency control: theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, December 1983. ISSN 0362-5915. 2.1.5

[14] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 729–738, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. 2.1.5.1

[15] Oracle. JDBC overview, November 2011. URL `http://www.oracle.com/technetwork/java/javase/jdbc/index.html`. 2.1.6

[16] Oracle. JDBC architecture, November 2011. URL `http://docs.oracle.com/javase/tutorial/jdbc/overview/index.html`. 2.1.6.1

[17] Interview questions java. JDBC driver types, November 2011. URL `http://www.jdbc-tutorial.com/jdbc-driver-types.htm`. 2.1.6.2

[18] Wikipedia. JDBC driver, November 2011. URL `http://en.wikipedia.org/wiki/JDBC_driver`. 2.1.6.2

[19] Wikipedia. ODBC, November 2011. URL `http://en.wikipedia.org/wiki/ODBC`. 2.1.6.2

[20] K. Driscoll, B. Hall, Håkan Sivencrona, and P. Zumsteg. Byzantine fault tolerance, from theory to reality. In Stuart Anderson, Massimo Felici, and Bev Littlewood, editors, *SAFECOMP*, volume 2788 of *Lecture Notes in Computer Science*, pages 235–248. Springer, 2003. ISBN 3-540-20126-2. 2.2.2

[21] L. Lamport. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982. 2.2.2

[22] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990. ISSN 0360-0300. 2.2.2.1

[23] Dalia Malki and Michael Reiter. A high-throughput secure reliable multicast protocol. In *Proceedings of the 9th IEEE workshop on Computer Security Foundations*, CSFW '96, pages 9–, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7522-5. 2.2.2.3

[24] Michael K. Reiter. Secure agreement protocols: reliable and atomic group multicast in rampart. In *Proceedings of the 2nd ACM Conference on Computer and communications security*, CCS '94, pages 68–80, New York, NY, USA, 1994. ACM. ISBN 0-89791-732-4. 2.2.2.3

[25] Ilir Gashi Nuno Neves Miguel Garcia, Alysson Bessani and Rafael Obelheiro. OS diversity for intrusion tolerance: Myth or reality? *Dependable Systems and Networks, International Conference on*, 0:383–394, 2011. 2.2.3, 3.4.1

[26] Ran Canetti, Rosario Gennaro, and Amir Herzberg. Proactive security: Long-term protection against break-ins. *CryptoBytes*, 3:1–8, 1997. 2.2.3

[27] Paul E. Ammann and John C. Knight. Data diversity: An approach to software fault tolerance. *IEEE Trans. Comput.*, 37(4):418–425, April 1988. ISSN 0018-9340. 2.2.3.1

[28] Ilir Gashi and Peter Popov. Rephrasing rules for off-the-shelf sql database servers. In *Proceedings of the Sixth European Dependable Computing Conference*, EDCC '06, pages 139–148, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2648-9. 2.2.3.1

[29] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, April 1980. ISSN 0004-5411. 2.3.1

[30] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2000. ISBN 0201657686. 2.3.1

[31] João Sousa and Alysson Bessani. From Byzantine Consensus to BFT State Machine Replication: A latency-optimal transformation. In *Proc. of the 9th European Dependable Computing Conference*, EDCC'12, Sibiu, Romania, May 2012. 3.1, 3.4.1

[32] Hector Garcia Molina, Frank Pittelli, and Susan Davidson. Applications of Byzantine agreement in database systems. *ACM Trans. Database Syst.*, 11(1):27–47, March 1986. ISSN 0362-5915. 3.2.2.2

[33] Tansaction Processing Performance Council. Tpc-c, April 2012. URL `http://www.tpc.org/tpcc/`. 4

[34] MySQL. MySQL, April 2012. URL `http://www.mysql.com`. 4.2

[35] PostgreSQL. PostgreSQL, April 2012. URL `http://www.postgresql.org/`. 4.2

[36] PostgreSQL. PostgreSQL Documentation: SQL Conformance, April 2012. URL `http://www.postgresql.org/docs/9.1/static/features.html`. 4.2

[37] HyperSQL. HSQLDB, April 2012. URL `http://hsqldb.org/`. 4.2

[38] HyperSQL. Chapter 2: SQL Language, April 2012. URL `http://hsqldb.org/doc/2.0/guide/sqlgeneral-chapt.html`. 4.2

[39] Firebird Project. FirebirdSQL, April 2012. URL `http://www.firebirdsql.org/`. 4.2

[40] BenchmarkSQL. BenchmarkSql, April 2012. URL `http://benchmarksql.sourceforge.net/index.html`. 4.2

[41] Red Hat. Hibernate, April 2012. URL `http://www.hibernate.org/`. 5